MoVAL : Modélisation multipoints de vue /multi-granularités d'architectures logicielles

Ahmad Kheir

Thèse de Doctorat

Ahmad KHEIR

Mémoire présenté en vue de l'obtention du
grade de Docteur de l'Université de Nantes
Docteur de L'Université Libanaise
sous le label de l'Université de Nantes Angers Le Mans

École doctorale : Sciences et technologies de l'information, et mathématiques

Discipline : Informatique et applications, section CNU 27
Unité de recherche : Laboratoire d'informatique de Nantes-Atlantique (LINA)

Soutenue le 13 Novembre 2014

MoVAL : Modélisation multipoints de vue /multi-granularités d'architectures logicielles

JURY

Président :	**M. Bilal CHEBARO**, Professeur, Université Libanaise
Rapporteurs :	**M^me^ Parisa GHODOUS**, Professeur, Université de Lyon I
	M. Henri BASSON, Professeur, Université du littoral cote d'opale
Examinateurs :	**M. Bilal CHEBARO**, Professeur, Université Libanaise
	M. Jean Claude ROYER, Professeur, Ecole des mines de Nantes
Directeurs de thèse :	**M. Mourad OUSSALAH**, Professeur, Université de Nantes
	M. Kifah TOUT, Professeur, Université Libanaise
Co-directrice de thèse :	**M^me^ Hala NAJA**, Maître de conf., Université Libanaise

Remerciements

1

Introduction

Le travail de cette thèse se situe dans le domaine des architectures logicielles, et porte sur la modélisation d'une architecture logicielle multipoints de vue et multi-granularités pour des systèmes informatiques évolutifs.

1.1 Qu'est ce qu'une architecture ?

Le terme architecture est issu d'un mot grec qui signifie "chef" ou "principe".
Depuis longtemps, la notion d'architecture est liée à la construction des édifices et des bâtiments complexes comme les pyramides pharaoniques, l'église La Sagrada Familia, et beaucoup d'autres monuments architecturaux que nous avons connus dans notre histoire. Le dictionnaire Larousse [Larousse, 1995] définit l'architecture d'un édifice comme étant sa finalité : "*La finalité de tout édifice est la réalisation d'un lieu qui isole ses occupants tout en ménageant des échanges (locomoteurs, thermiques, optiques) avec le milieu extérieur. Le type de l'édifice (parti, matériaux, structure, éventuel décor) est conditionné par les ressources techniques de chaque civilisation confrontées aux conditions physiques du lieu et par le programme (destination) qui lui est assigné*".

1.2 Les architectures logicielles en informatique

Depuis la fin des années 1960s, les informaticiens ont emprunté et adapté la notion d'architecture pour l'appliquer dans la construction des systèmes informatiques ; ainsi on a vu apparaître la notion d'architecture logicielle [Naur and Randell, 1969].
Une architecture logicielle en informatique décrit les différents éléments constituant un ou plusieurs systèmes informatiques, leur organisation, interrelation, et interactions. L'architecture d'un système informatique est l'ensemble de ces concepts et propriétés fondamentaux dans son environnement, incarnés dans ses éléments, relations, et les principes de sa conception et son évolution.

1.3 Les apports des vues en architecture logicielle

Tout système informatique, qu'il soit simple ou difficile à comprendre, est constitué de plusieurs éléments liés ensemble. Il peut avoir un petit nombre d'éléments, ou peut-être un seul élément, comme il peut avoir une dizaine ou une centaine d'éléments ; cette liaison peut être triviale ou complexe.

En plus, tout système informatique est constitué de plusieurs éléments qui interagissent l'un avec l'autre et avec l'environnement externe au système d'une manière déterministe ou prévisible. Le comportement de ces éléments peut être simple et facile à comprendre, et il peut être de même très compliqué pour qu'il soit compris entièrement par une seule personne. Ce comportement reste toujours existant pour tout système informatique, qu'il soit documenté ou non. En d'autres termes, tout système informatique dispose d'une architecture, qu'elle soit documentée dans une description d'architecture ou non.
L'introduction des vues en architecture logicielle a contribué à améliorer en plus le processus de description d'une telle architecture de plusieurs manières [Rozanski and Woods, 2011] :

- la séparation des préoccupations : la séparation des multiples aspects d'un système en plusieurs modèles différents aide durant le processus d'analyse, de conception, etc., en permettant de se focaliser dans chaque étape à un aspect particulier ;

- la communication avec les groupes d'intervenants : la communication entre des groupes d'intervenants avec des préoccupations diverses est un défi pour l'architecte. Les approches basées sur les vues offrent la possibilité aux intervenants de converger assez rapidement vers les descriptions d'architecture qui les intéressent répondant à leurs préoccupations ;

- la gestion de la complexité : considérer simultanément dans un même modèle tous les aspects d'un système induit une complexité qu'un être humain ne peut pas gérer. Le fait de décomposer le modèle en des modèles selon des vues différentes, réduit remarquablement la complexité.

1.4 Les limites des approches existantes

Malgré les avantages apportés par les approches d'architecture logicielle multipoints de vue en termes de séparation des préoccupations des systèmes informatiques et de réduction de la complexité, ces dernières souffrent cependant encore de plusieurs limitations, comme :

- la complexité inhérente au sein d'une vue. En effet, les préoccupations primitives de très haut niveau associées à une vue particulière d'une architecture logicielle ne peuvent jamais être traitées directement dans un seul modèle associé à un seul niveau d'abstraction, qui contient tous les détails nécessaires pour mener l'implémentation. Pourtant, on a besoin toujours de développer plusieurs modèles dans plusieurs niveaux d'abstraction pour pouvoir répondre effectivement à ces préoccupations primitives. Ainsi, la présence d'une hiérarchie plate d'une vue architecturale dans laquelle coexistent des modèles de très haut niveau avec des modèles très proches de l'implémentation est toujours inconfortable et déroutante pour les architectes logiciels ;

- les problèmes d'inconsistance qui peuvent avoir lieu entre les différentes vues de l'architecture. Ces problèmes d'inconsistance résultent du fait que les préoccupations du système informatique seront traitées séparément dans plusieurs vues de l'architecture logicielle, sans que les intervenants impliqués dans la construction d'une vue donnée de cette architecture, seront impliqués dans la construction des autres vues ;

- l'absence d'un processus de définition de ces architectures logicielles qui soit complet et adaptable avec les processus de dÃ©veloppement logiciels. En effet, la tâche de définition d'architectures logicielles est une tâche vraiment lourde, critique et difficile. Pour cela, un architecte logiciel a toujours besoins de meilleures pratiques (mieux connues en anglais par *best practices*) rassemblées dans une méthodologie ou un processus de définition complet, afin qu'il soit capable de construire effectivement, et dans une durée acceptable, une architecture logicielle appropriée répondant aux problèmes visés par le système informatique. Normalement, ce processus doit être adaptable avec les différents processus de développement qui peuvent être adoptés durant la construction du système informatique.

1.5 La contribution de la thèse

Dans cette thèse, nous proposons une nouvelle approche d'architecture logicielle multipoints de vue, dénotée *MoVAL* (*Model, View, and Abstraction Level based software architecture*), qui est basée sur le standard *ISO/IEC/IEEE 42010*, intitulé "*Systems and software engineering - Architecture description*" [ISO/IEC/IEEE, 2011]. Cette approche répond aux limites des approches existantes, et elle est caractérisée par les points suivants :

- *MoVAL* est une approche d'architecture logicielle multipoints de vue ;
- Elle définit pour chaque vue une hiérarchie de niveaux d'abstraction ;
- Les niveaux d'abstraction d'une vue dans *MoVAL* sont de deux types : (1) les niveaux de réalisation d'une vue et (2) les niveaux de description associés à chaque niveau de réalisation ;
- *MoVAL* définit différents types de liens architecturaux assurant la cohérence de l'architecture et la résolution des inconsistances affrontées ;
- Dans l'approche *MoVAL*, un processus de définition d'architecture est présenté pour guider la construction des architectures logicielles.

1.6 Le plan de la thèse

Ce mémoire de thèse est articulé autour de trois parties distinctes :

- La première partie intitulée "État de l'art" comporte deux chapitres :
  - le chapitre "Synthèse bibliographique" (Chapitre 2) présente une synthèse bibliographique à propos de l'application de la notion de vue/point de vue dans quatre domaines de l'informatique : (1) la spécification des besoins, (2) la modélisation des systèmes, (3) la programmation, finalement et surtout en (4) architecture logicielle.
  - le chapitre "Analyse comparative et limitations" (Chapitre 3) présente une analyse comparative entre les différentes approches d'architecture logicielle multipoints de vue afin de déterminer leurs défaillances majeures et leurs limitations.
- La deuxième partie de cette thèse, intitulée "MoVAL", présente notre contribution dans le domaine et comporte trois chapitres :
  - le chapitre "MoVAL : Approche et concepts de base" (Chapitre 4) présente les motivations de notre approche, et définit ses éléments de base. Ensuite, plusieurs types de visualisation d'une architecture logicielle et un catalogue de point de vue seront proposés.
  - dans le chapitre "Formalisation de MoVAL" (Chapitre 5) nous formalisons les définitions présentées dans le chapitre précédents afin d'assurer la cohérence et la complétude de notre proposition.
  - le chapitre "MoVAL-ADP : le processus de définition d'architecture adapté à MoVAL" (Chapitre 6) présente un processus de définition d'architecture logicielle spécifique à *MoVAL*, dénoté *MoVAL-ADP*, qui est conforme au processus unifié (*UP*) et divisé en quatre phases principales : la phase *Inception*, la phase *Elaboration*, la phase *Construction* et la phase *Transition*.
- La troisième partie de cette thèse intitulée "Expérimentations", se concentre dans son unique chapitre sur la validation pratique de l'approche. Ainsi, nous présentons dans le chapitre 7 le prototype d'implémentation de notre outil "*MoVAL-Tool*".

Finalement, le chapitre 8 conclut ce mémoire et présente nos contributions et nos perspectives.

État de l’art

2

Synthèse bibliographique

2.1 Introduction

La notion de point de vue (appelée aussi vue, perspective ou *viewtype*) revêt des significations diverses suivant les domaines et les travaux en informatique. En général, on s'intéresse aux points de vue dès lors que l'on modélise des systèmes à grande échelle faisant intervenir une grande masse de données, nécessitant la coopération de plusieurs experts de différents domaines de connaissances et points d'intérêts et s'adressant à une vaste panoplie d'utilisateurs. Comme souligné dans [Naja, 1998], un point de vue met en liaison un concepteur, un univers de discours (c'est-à-dire le système à modéliser) et l'objectif que le concepteur cherche à réaliser. Le point de vue permet une représentation partielle du système à modéliser mettant en relief un ou plusieurs aspects de ce dernier et occultant d'autres.
Les premiers travaux sur les points de vue s'inscrivent dans le domaine de la représentation des connaissances en Intelligence artificielle. Ici on peut citer les travaux de *Minsky* [Minksy, 1975] et Lee et al. [Erman and Lesser, 1975] en 1975 suivis des langages de programmation *KRL* [Bobrow and Winograd, 1977], *LOOPS* [Bobrow and Stefik, 1983] et *ROME* [Carré et al., 1990] qui ont tous mis en évidence la nécessité de conférer à un même objet plusieurs représentations.
En Bases de données, la notion de vue a été introduite pour la première fois dans le rapport *d'Ansi/Sparc* [Klug and Tsichritzis, 1977] sous le nom de schéma externe. Un schéma externe représente le seul point d'accès possible pour les utilisateurs, aux données de la base. C'est une portion de la base intéressant un usager ou un groupe d'usagers et intégrant la notion de droits d'accès. Une vue restreint la visibilité des données et/ou adapte leur structure aux besoins d'une application. Elle permet de résoudre des problèmes liés à la configuration de l'interface utilisateur, la protection des données, l'évolution des besoins d'organisation des données sans perturbation de l'existant et l'optimisation des requêtes.
En Génie logiciel, la motivation des points de vue est la séparation des préoccupations (c'est-à-dire *Separation of concerns*). Ainsi, les vues sont introduites comme des éléments de construction pour la gestion des complexités des artefacts produits (comme les spécifications des besoins, les modèles de conception et les programmes). Dans ce qui suit, nous effectuons une synthèse bibliographique sur les vues dans quatre domaines du génie logiciel qui sont : la spécification des besoins, la modélisation de systèmes, l'implémentation de systèmes et l'ingénierie des architectures logicielles. Nous nous étendons sur les travaux en architecture logicielle car ils sont en étroite liaison avec l'objet de notre étude.

2.2 Les vues en spécification des besoins

La spécification des besoins, ou *Software Requirements Specification* (*SRS*), est définie dans le standard IEEE 830-1998 [Board, 1998] comme étant une description du comportement d'un système informatique et les interactions que les utilisateurs peuvent avoir avec ce système. En plus, la spécification des besoins contient les besoins non-fonctionnels imposés par le client comme la performance, la qualité, les contraintes, etc. ... .
Normalement, deux phases peuvent être distinguées en spécification des besoins, comme l'ont souligné Ross et Schoman [Ross and Schoman Jr, 1977] :

- L'analyse des besoins, qui porte sur la réalisation d'un énoncé textuel des besoins du client, qui est en général ambigu et manque de précisions.
- La définition des besoins, dans laquelle des langages spécifiques pour exprimer ces besoins peuvent être utilisé afin d'avoir un bon cahier des charges caractérisé par les huit qualitatifs suivants, présentés dans le standard IEEE 83-1998 :
  - Correct : un cahier des charges est correct si et seulement si tous ses besoins sont des besoins qui doivent être implémentées dans le système ;
  - Non-ambigu : un cahier des charges est non-ambigu si et seulement si chacune de ses besoins possède une seule interprétation ;
  - Complet : un cahier des charges est complet si et seulement si, il contient tous les besoins fonctionnels et non-fonctionnels du système, une définition de toutes les réponses du système dans les situations valides et non-valides, toutes les références sur les figures, tableaux, diagrammes, et unités de mesure utilisées ;
  - Consistent : un cahier des charges est consistent si et seulement s'il ne contient jamais des contradictions entre les besoins qu'il définit et avec les besoins définies dans des documents de plus haut niveau ;
  - Classé pour importance et/ou stabilité : un cahier des charges est classé pour importance et/ou stabilité si chacune de ses besoins possède un identificateur indiquant son niveau d'importance ou de stabilité ;
  - Vérifiable : un cahier des charges est vérifiable si et seulement si, pour chacune de ses besoins, il existe un processus bien défini et effectif permettant à une personne ou une machine de vérifier si le système satisfait cette besoin ;
  - Modifiable : un cahier des charges est modifiable si et seulement si n'importe quel changement sur ses besoins peut être inclut sans affecter son style et structure ;
  - Traçable : un cahier des charges et traçable si l'origine de chacune de ses besoins est bien clair, et s'il facilite le référencement de ses besoins dans les documentations futures.

Dans le domaine de spécification des besoins plusieurs approches à base de points de vue ont été proposées depuis les années 1970, comme les travaux de *Ross et al.*, l'approche *CORE* de *Mullery*, les approches à base de dialogue, les approches à base d'heuristiques, les travaux de *Delugach*, et finalement l'approche *Preview* de *Sommerville*.

2.2.1 Les travaux de *Ross et al.*

Ross et al. ont été parmi les premiers qui ont distingué dans leurs travaux, en 1977, entre l'analyse des besoins d'un système informatique et la définition de ces besoins, avant que le standard *IEEE 830-1998* adopte cette distinction en 1993 et fournit à sa base un standard pour la spécification des besoins. Alors pour *Ross et al.* la définition de besoins représente un processus complet, beaucoup plus précis et beaucoup plus organisé visant à :

- **L'analyse du contexte**, afin de savoir pourquoi un tel système doit être créé selon certains critères de faisabilité techniques, opérationnels, et économiques ;
- **La spécification fonctionnelle**, décrivant que doit ce système offrir en terme de fonctionnalités ;
- **Les contraintes de conception**, récapitulant les conditions spécifiant comment ce système doit être construit et implémenté.

Le résultat de la définition/spécification des besoins est l'obtention d'une architecture fonctionnelle qui représente les spécifications fonctionnelles liées d'une part au contexte du système, et d'autre part aux contraintes de conception de ce système. Ainsi, une nouvelle proposition a été présentée, qui se base sur la construction de cette architecture fonctionnelle suivant différents points de vue qui représentent les aspects à considérer et qui doivent être accomplis.

2.2.2 L'approche *CORE*

G. P. Mullery a développé une méthode formelle pour la spécification des besoins et la modélisation des systèmes informatiques complexes. Dans cette méthode, appelée *CORE* (*Controlled Requirement Expression*) [Mullery, 1979], une décomposition du processus de modélisation en plusieurs étapes a été proposée. Cette décomposition se base sur l'identification, pour chaque étape, des points de vue associés afin d'avoir finalement une représentation schématisée du système en tenant compte de tous les points de vue de ce système et les chevauchements qui peuvent avoir lieu entre eux. *CORE* a été parmi les premières approches dans ce domaine enrichies par une notation graphique basée sur deux types de diagrammes, les diagrammes à processus unique (*"Single-thread" diagrams*) et les diagrammes opérationnels (*"Operational" diagrams*) afin de soutenir les cas de parallélisme et les itérations.
Le processus de spécification des besoins et de modélisation qui a été défini dans l'approche CORE est basé sur trois activités principales :

1. proposer les points de vue pertinents,
2. définir les points de vue pertinents,
3. confirmer les points de vue pertinents.

Dans la première activité, le but sera de proposer deux à cinq points de vue pertinents pour le niveau de décomposition actuel (*i.e.* l'itération actuelle) en se basant sur une discussion, à propos des conseils techniques et des informations obtenus dans les itérations passées, avec le client et les représentants des utilisateurs. Puis dans la deuxième étape, chaque point de vue doit être défini à travers les diagrammes et la notation graphique proposée, suite à des interviews menés par les analystes avec les utilisateurs et les experts techniques associés. Finalement, la troisième activité consiste d'une part à la combinaison des diagrammes des points de vue individuels, construits dans la deuxième activité, pour obtenir des diagrammes plus complets, et d'autre part la vérification de la fiabilité de ces diagrammes.

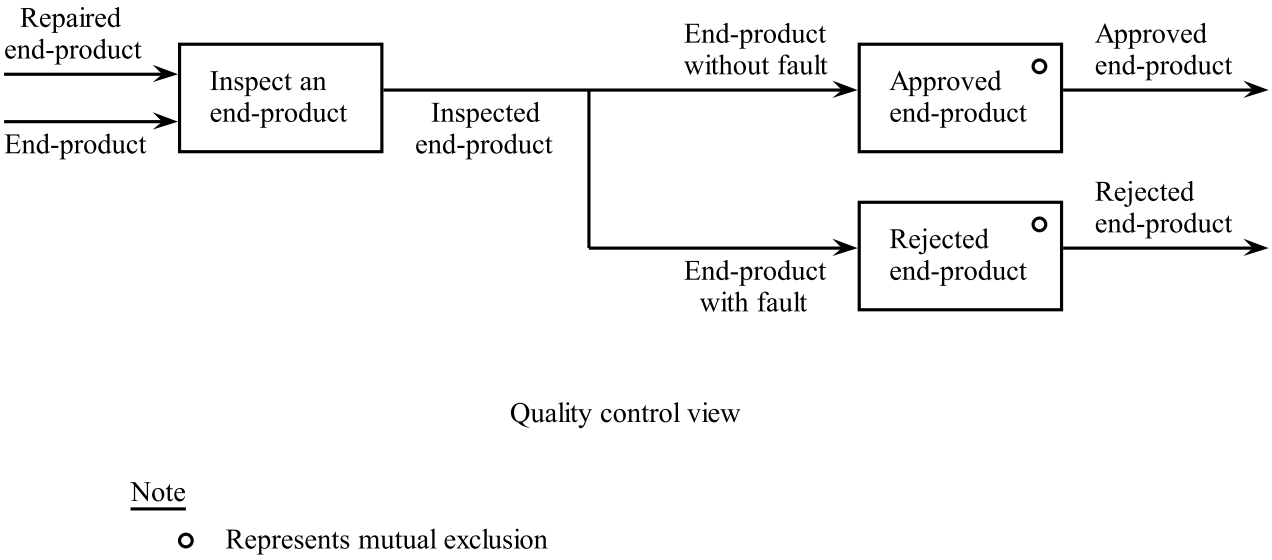


FIGURE 2.1 – Exemple d'un diagramme construit suivant la notation graphique de *CORE* extrait de [Mullery, 1979].

2.2.3 Les approches à base de dialogue

En spécification des besoins, plusieurs approches basées sur les notions de dialogue et de négociations sont apparues. Parmi ces approches on peut citer le modèle de spécification des besoins à partir de plusieurs points de vue, qui a été proposé par *A. Finkelstein* et *H. Fuks* en 1989 [Finkelstein and Fuks, 1989]. Dans cette approche, le développement des spécifications des besoins se fait à travers un dialogue qui s'établit entre plusieurs points de vue afin d'établir les responsabilités menant à construire le modèle de spécification des besoins.
En effet, le processus proposé dans [Finkelstein and Fuks, 1989] est divisé en deux parties distinctes :

- la construction d'une architecture de points de vue,
- la construction d'un plan de dialogue animé par l'architecture des points de vue.

Dans cette approche un point de vue est défini comme étant un agent qui est responsable de maintenir une perspective particulière, c'est un participant logique présenté par ou installé sur un autre participant physique. Aussi, la structure d'un point de vue composée de plusieurs unités est définie comme suit :

- l'unité "*Commitment store*" qui sauvegarde tous les engagements de ce point de vue. Normalement la liste de ces engagements ne peut jamais être changée que suite à un dialogue avec un autre participant physique ;
- l'unité "*Working area*", c'est la base de données du point de vue qui contient les déclarations qui n'ont pas été encore publiées sous forme d'engagements ;
- l'unité "*Event store*" qui maintient des records sur les évènements des dialogues établis. Ces informations vont être utilisées par la quatrième unité, le "*Dialogue kernel*" ;
- l'unité "*Dialogue kernel*", c'est l'unité la plus importante dans la structure d'un point de vue, puisqu'elle est le contrôleur indépendant de ce dernier responsable d'implémenter la stratégie associée.

La figure 2.2 illustre la structure interne proposée dans [Finkelstein and Fuks, 1989] pour un point de vue.

Afin d'assurer la consistance de l'approche, les besoins sont exprimées dans [Finkelstein et al., 1993] par des formules logiques permettant la validation de l'ensemble des besoins du système informatique.
En 1990, un outil dénoté *Oz* [Robinson, 1990] est proposé, dans le même cadre de spécification des besoins

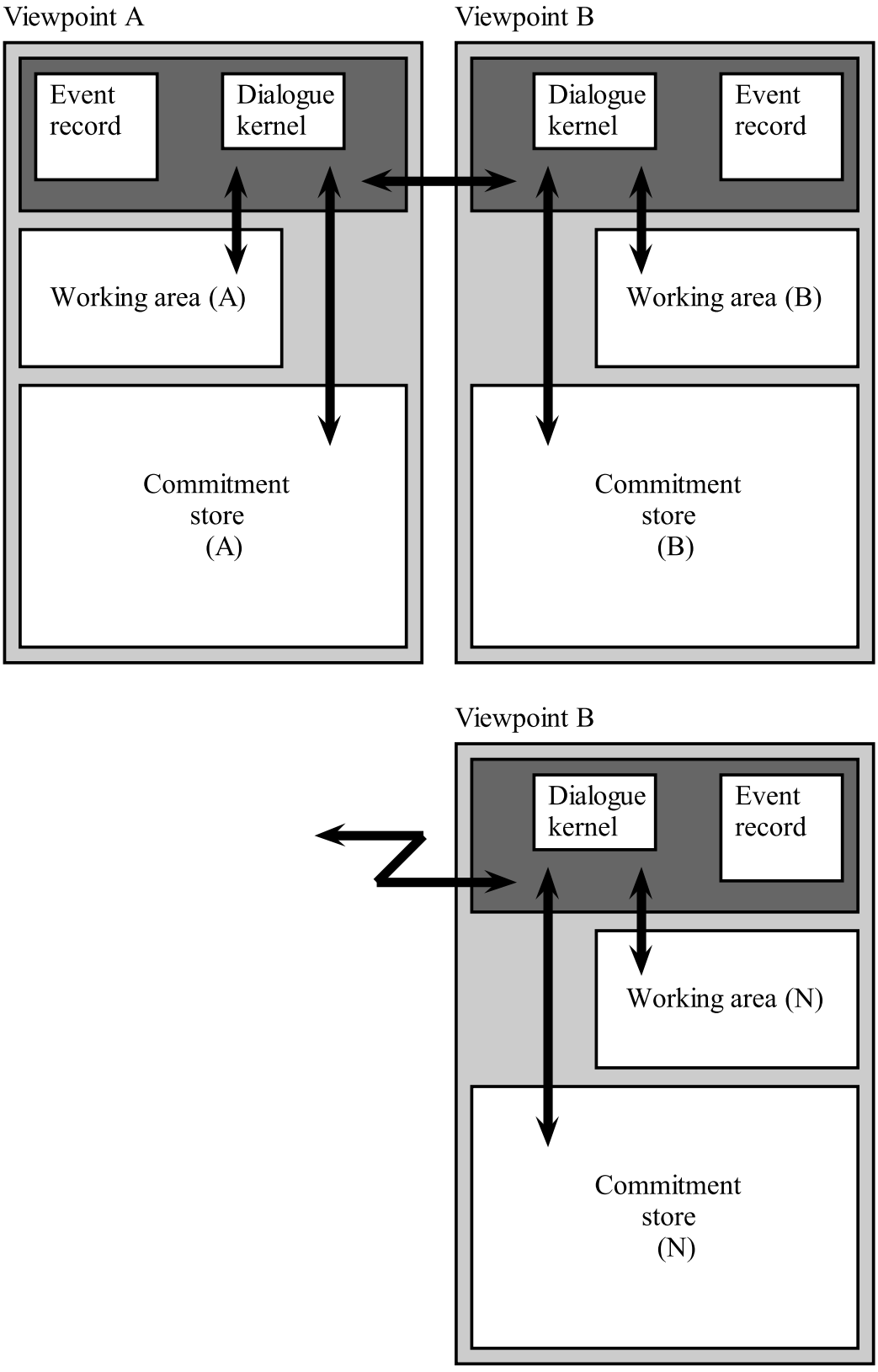


FIGURE 2.2 – Structure interne d'un point de vue, extrait de [Finkelstein and Fuks, 1989].

à partir de la notion de négociation. Cette approche adresse la spécification semi-automatique des besoins d'un système informatique à travers la représentation des perspectives des différents utilisateurs du système et des analystes par des agents logiciels. Ces agents vont négocier, chacun depuis la perspective qu'il représente, les spécifications des besoins pour avoir finalement un modèle formel des intentions des utilisateurs vis-à-vis du système, et les méthodes de résolution des conflits qui ont eu lieu entre ces différentes perspec-

tives durant la négociation.
Aussi, dans [Niskier et al., 1989], qui utilise la notion des points de vue afin d'organiser les heuristiques qui vont être adoptées ensuite par des agents afin de construire une spécification complète des besoins du système informatique. Dans [Sampaio do Prado Leite and Freeman, 1991], *Leite et al.* exprime les besoins à travers des heuristiques groupées dans des vues. Ces heuristiques seront ensuite utilisées afin de valider les besoins acquis.

2.2.4 Plateforme pour la spécification des besoins à base de points de vue

En 1991, une nouvelle approche a été proposée par *A. Finkelstein et al.* [Finkelstein et al., 1991], cette approche représente une plateforme qui organise la structure du système logiciel et le processus de développement logiciel à travers la notion de point de vue. Ainsi, selon cette approche, un point de vue capture un rôle particulier et les responsabilités prises en charge par un participant particulier pendant une phase donnée du processus de développement. En plus, le point de vue doit encapsuler seulement les aspects du domaine d'application pertinents à son rôle, et doit utiliser un style unique et bien défini pour représenter les connaissances imbriquées.
D'une autre part, les types des points de vue ont été définis pour permettre à deux points de vue distincts de capturer les intérêts associés à deux parties différentes du système à partir de la même perspective. Finalement, des relations ont été définies qui peuvent avoir lieu entre deux points de vue soit appartenant au même domaine d'application, ou associé au même type de point de vue.

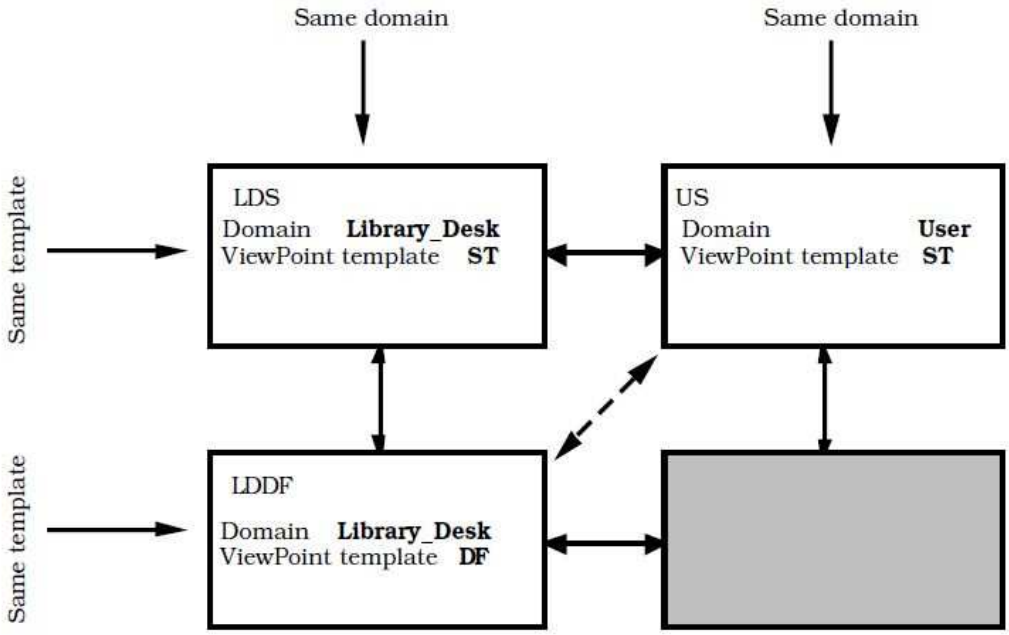


FIGURE 2.3 – Les relations entre les points de vue, extraite de [Finkelstein et al., 1991].

Cette approche a été ensuite étendue dans [Nuseibeh et al., 1994, Easterbrrok and Nuseibeh, 1996] par *Nuseibeh et al.*. Ainsi, les relations entre les différents points de vue d'un système ont été définies plus précisément et plus rigoureusement afin de maintenir la consistance de ce système après que l'on a décomposé en plusieurs entités ou plusieurs points de vue, et afin d'assurer la communication et le transfert de données entre ces derniers.

2.2.5 Les travaux de Delugach

Depuis les débuts des années 1990s, *Harry S. Delugach* et ses collègues ont effectué beaucoup de travaux dans le domaine de la spécification des besoins des systèmes logiciels. Ces travaux ont commencé

dans [Delugach, 1990, Delugach, 1991] pour développer une approche d'analyse des besoins à base de points de vue, utilisant les graphes conceptuels. Dans ce cadre, les différentes perspectives des clients, des concepteurs, et de tout autre participant dans le processus de développement du système logiciel ont été prises en compte. Ainsi, les besoins de chaque participant, exprimés par une notation ou un formalisme particulier seront transformés en un graphe conceptuel. Puis, les conflits qui peuvent avoir lieu entre les différentes représentations des besoins des différents participants seront détectés pour les régler, et faire enfin la combinaison de tous les graphes obtenus afin d'avoir une représentation globale des besoins de tous les participants dans un seul graphe conceptuel.
En 1992, [Delugach, 1992], cette approche a été enrichie en définissant quatre modèles de traduction de quatre notations différentes en graphe conceptuel. Ces quatre notations sont :

(1) les diagrammes Entité-Relation (*Entity-Relationship diagrams*),

(2) les diagrammes de flux de données (*Data Flow diagrams*),

(3) les diagrammes d'état-transition (*State Transition diagrams*),

(4) réseaux de besoins (*Requirements Networks*).

Afin d'assurer une meilleure cohérence et consistance entre les différents graphes conceptuels et les différents points de vue, un environnement plus complet et plus rigoureux de spécification des besoins a été mis en oeuvre dans [Delugach, 1996], en développant le concept de validation des besoins acquis pour chaque point de vue, de détection et d'analyse formelle des relations qui peuvent éventuellement exister entre les différentes vue. La figure 2.4 illustre l'environnement de spécification des besoins proposé.

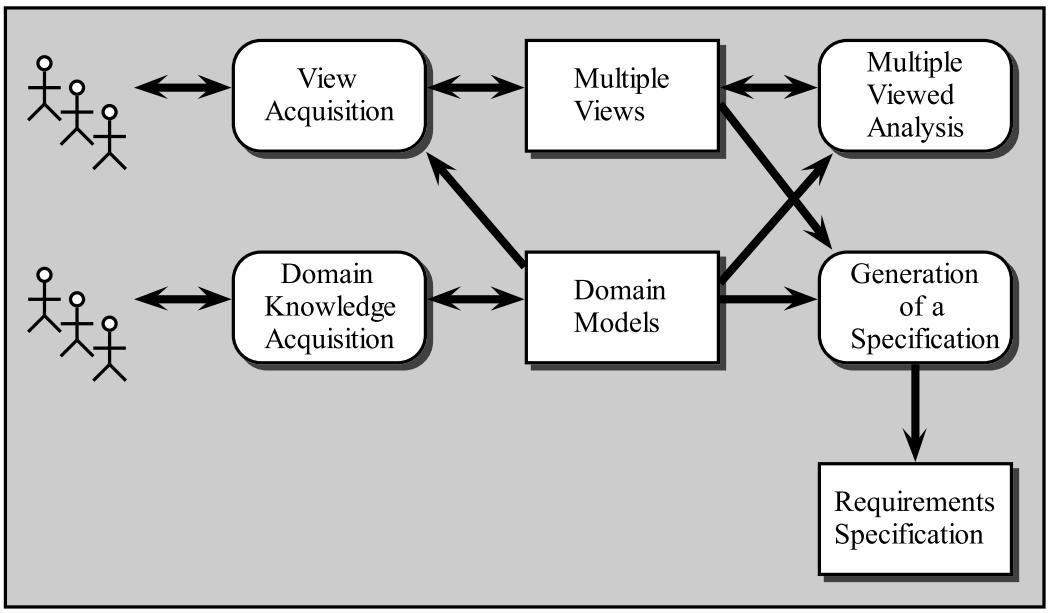


FIGURE 2.4 – Environnement de spécification des besoins multi-points de vue, extrait de [Delugach, 1996].

2.2.6 L'approche *Preview*

Dans [Sommerville and Sawyer, 1997] une étude bibliographique sur les points de vue et leurs utilisations dans les différentes approches de spécification des besoins a été munie, en donnant les points forts, les faiblesses, et les problèmes rencontrés pour chaque utilisation. Après, une nouvelle approche de spécification des besoins, dénotée *Preview* (*Process and Requirements Engineering Viewpoint*), a été introduite.
Dans cette approche, trois classes de points de vue ont été définies, associées à :

(1) une personne ou une machine qui interagit directement avec le système ;

(2) un intervenant indirect qui a des intérêts dans le système mais sans avoir besoin d'interagir avec lui ;

(3) un ensemble de caractéristiques du domaine.

En plus, cette approche définit des préoccupations qui représentent des objectifs stratégiques du système, et qui doivent être toujours respectées dans les différents points de vue. La figure 2.5 représente la relation entre les points de vue et leurs préoccupations.

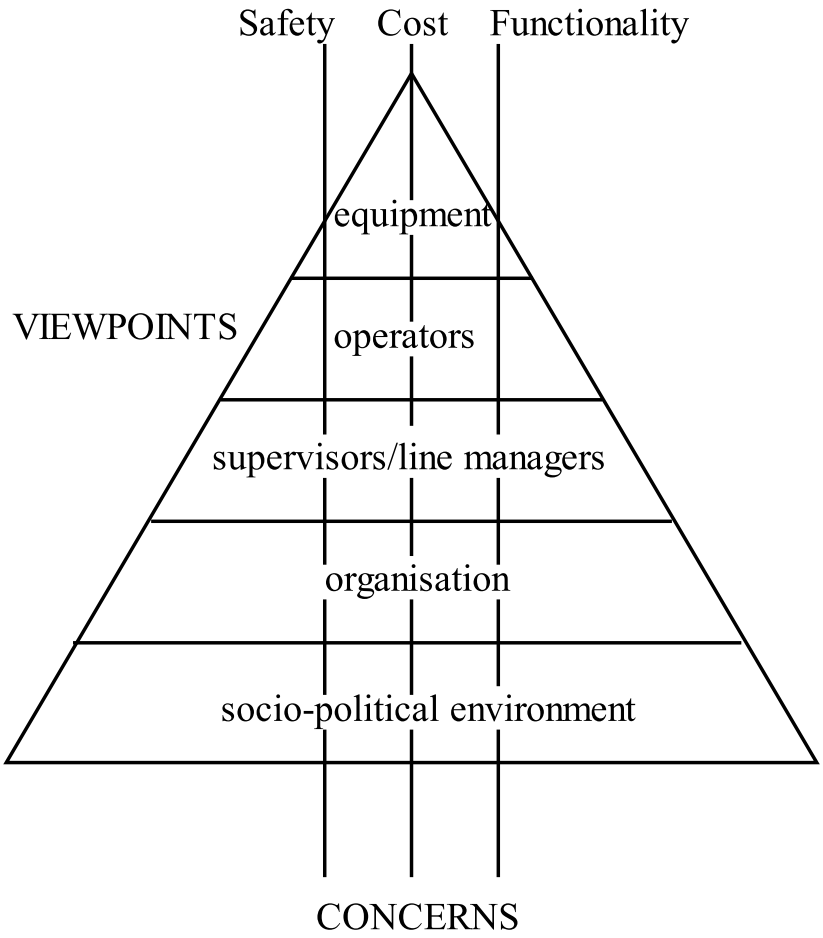


FIGURE 2.5 – L'orthogonalité des points de vue et des préoccupations dans *Preview*, extraite de [Sommerville and Sawyer, 1997].

Enfin, un processus informel qui peut être utilisé pour appliquer l'approche Preview a été proposé. Ce processus consiste en plusieurs étapes illustrées dans la figure 2.6.

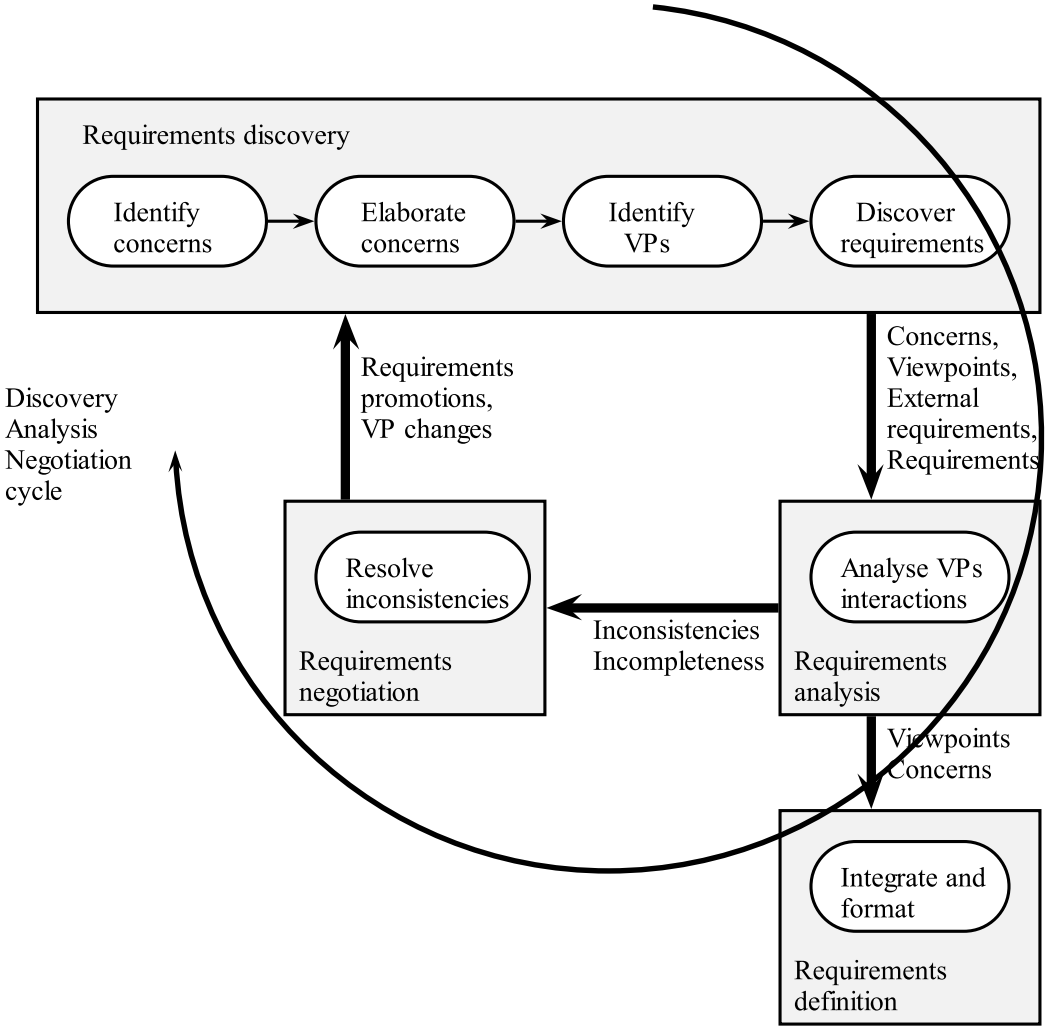


FIGURE 2.6 – Le processus proposé dans *Preview*, extrait de [Sommerville and Sawyer, 1997].

2.3 Les vues en modélisation de systèmes

Pour définir le terme modélisation des systèmes informatiques, il est indispensable de définir a priori le terme conception des systèmes informatiques. Ainsi, la conception des systèmes est la définition de ses méthodes, fonctions, composants, structure globale, et l'interaction du code d'une manière à avoir comme résultats des fonctionnalités satisfaisantes aux besoins des utilisateurs du système. La conception d'un système informatique doit aussi inclure les matériaux qui doivent être utilisés (*i.e. Hardware*), les bases de données, les plateformes tierces que le système doit utiliser ou interagir avec, les interfaces de programmation que le code du système doit utiliser ou offrir. Alors, la conception des systèmes informatiques représente l'image globale de ce qui va être exécuté, où et comment toutes les parties vont interagir.
Revenons à la modélisation des systèmes informatiques, c'est la représentation et l'expression de la conception de ces systèmes à travers des modèles spécifiques et appropriés, comme par exemple les modèles *UML*. Parmi les approches de modélisation de systèmes informatiques à base de points de vue, on va mentionner dans cette section l'approche *CÈDRE* proposée par *H. Naja*, l'approche *VBOOM* d'*Andrade*, le standard *UML*, les travaux de *Dijkman*, et l'approche *VUML*.

2.3.1 L'approche CÈDRE

Dans [Naja, 1998], l'auteur présente une approche, dénotée CÈDRE, qui s'inscrit dans le cadre des travaux sur la conception et la modélisation de bases de données orientées objets. Dans cette approche l'auteur se

focalise sur l'aspect multipoints de vue, et qui considère que les objets doivent être manipulés par plusieurs experts et donc décrits selon différents points de vue. Ainsi, il définit cinq propriétés qu'un modèle à objets devrait posséder pour qu'il soit un modèle objet multivues. Ces propriétés sont : (1) une portée d'un point de vue à différents niveaux ; (2) la représentation multiple possède un référentiel ; (3) la représentation multiple est décentralisée ; (4) les représentations partielles s'échangent des informations entre elles ; (5) la représentation multiple est cohérente.

2.3.2 L'approche VBOOM

Le projet *VBOOM* [Kriouile, 1995] munie du langage *VBOOL* [Marcaillou et al., 1994] introduit la notion de vue et point de vue dans l'analyse et la modélisation des systèmes orientés objets et l'implémente à travers une structure d'héritage.
Dans [Andrade et al., 2004], une méthodologie d'acquisition de besoins et de conception est proposée. Cette méthodologie consiste à identifier les points de vue pertinents au système en construction, puis pour chaque point de vue, l'acquisition et la conception des besoins et des intérêts imbriqués. Ensuite, une comparaison entre les modèles de conception appartenant à des points de vue différents, et qui semblent en relation afin de détecter les inconsistances potentielles pour les résoudre et réintégrer les points de vue dans le système. La figure 2.7 présente la méthodologie proposée.

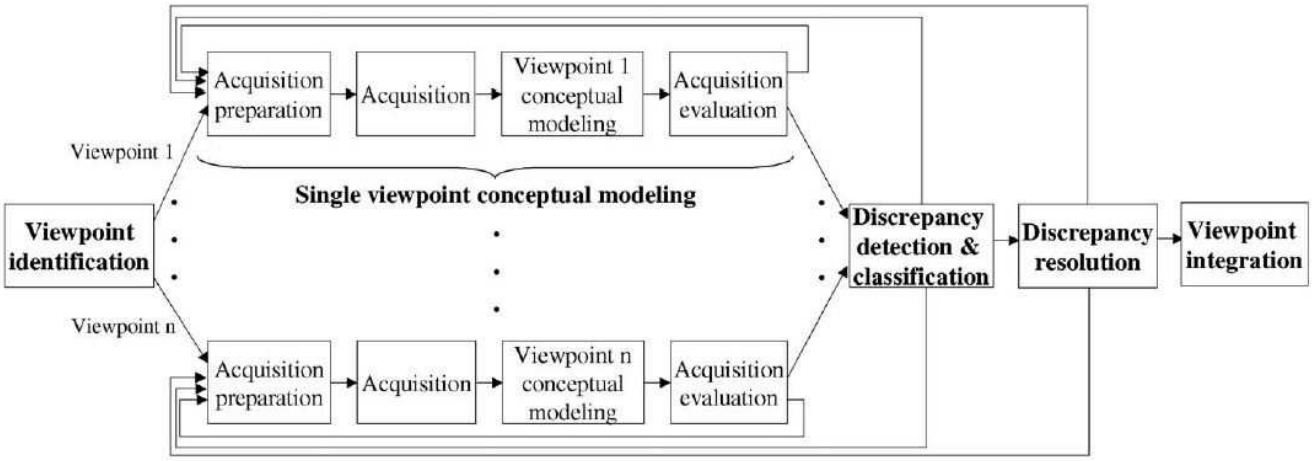


FIGURE 2.7 – La méthodologie de conception proposé par *Andrade et al.*, extraite de [Andrade et al., 2004].

2.3.3 Le standard *UML*

La notation *UML* [OMG, 2011] s'avère celle qui incarne le plus largement la notion de point de vue. En effet, *UML* propose dans sa norme actuelle treize types de diagrammes, où chaque type confère un point de vue implicite duquel un système complexe sera abordé, et permet ainsi de décliner le modèle complexe d'un système en plusieurs sous-modèles complémentaires moins complexes et plus facilement abordables et compréhensibles. Par exemple, le diagramme de classe cible l'aspect structurel du système informatique, tandis que les diagrammes de séquence, de collaboration et d'activité ciblent l'aspect comportemental de ce système. Egalement, *UML* propose un mécanisme d'extension qui permet aux utilisateurs d'ajouter ou de personnaliser les types de diagrammes prédéfinis et donc d'ajouter des points de vue.

2.3.4 Les travaux de *Dijkman*

Dans [Dijkman, 2006, Dijkman et al., 2008], la consistance entre les différents points de vue d'une architecture logicielle est étudiée. Alors, une plateforme qui maintient la consistance et la cohérence dans les structures multipoints de vue a été proposée. Ainsi, un point de vue est défini comme étant un regroupement des intérêts d'un intervenant particulier sous le niveau de détails, ou le niveau d'abstraction qui lui intéresse, et dans une position dans le processus de développement dans laquelle ce dernier intervient. Suivant cette approche, une vue sera représentée par un modèle graphique ou textuel construit suivant le patron défini dans le point de vue associé, à travers des concepts de base définis en utilisant *MOF* (*Meta-Object Facility*). En fait, la contribution la plus importante de cette approche réside dans la définition formelle et précise des relations et des règles de consistance entre les différents points de vue d'une structure logicielle. En effet, les règles de consistance ont été définies pour présenter les relations entre les concepts de base de différents points de vues en utilisant le langage de contrainte *OCL* (*Object Constraint Language*). Ces relations appartiennent très souvent à deux catégories principales :

(1) les relations de chevauchement (Overlap relations) qui existent entre des points de vue qui considèrent partiellement les mêmes préoccupations au même niveau d'abstraction ;

(2) et les relations de raffinement (Refinement relations) qui existent entre des points de vue qui considèrent partiellement les mêmes préoccupations mais à différents niveaux d'abstraction.

La figure 2.8 illustre un exemple d'ensemble de points de vue d'une structure logicielle.

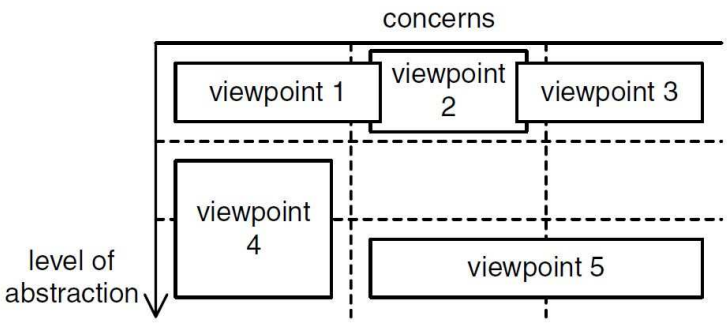


FIGURE 2.8 – Exemple de points de vue d'une structure logicielle, extrait de [Dijkman et al., 2008].

2.3.5 L'approche *VUML*

Aussi dans le cadre de la conception et de la modélisation des systèmes informatiques, une spécialisation d'*UML* a été développée et introduite dans [Nassar, 2003, Nassar et al., 2004], afin que ce dernier supporte la notion de vue et de point de vue.
Cette approche, nommée *VUML* (*View based Unified Modeling Language*), consiste à définir un diagramme de classe qui comporte les classes classiques définies dans *UML* et un autre type de classes multivues définies par deux stéréotypes, le stéréotype "*base*" et le stéréotype "*view*". le stéréotype *base* représente la partie commune accessible par tous les points de vue, tandis que le stéréotype *view* représente la partie spécifique du base associée à un point de vue particulier, et ces deux stéréotypes seront reliés par une relation de dépendance stéréotypée "*ViewExtension*". Un autre stéréotype de relations a été proposé et nommé "*ViewDependancy*" afin de maintenir la cohérence au sein d'une classe multivues, qui peuvent être exprimées en utilisant le langage *OCL* (*Object Constraint Language*). La figure 2.9 illustre un exemple de diagramme de classe multivues. De même, les composants multivues sont définis ; ils qui sont des composants classiques pour lesquels on peut attribuer des interfaces classiques et des autres multivues (*ViewInterface*) décrivant le comportement de ce composant et ses interactions avec l'extérieur en se basant sur

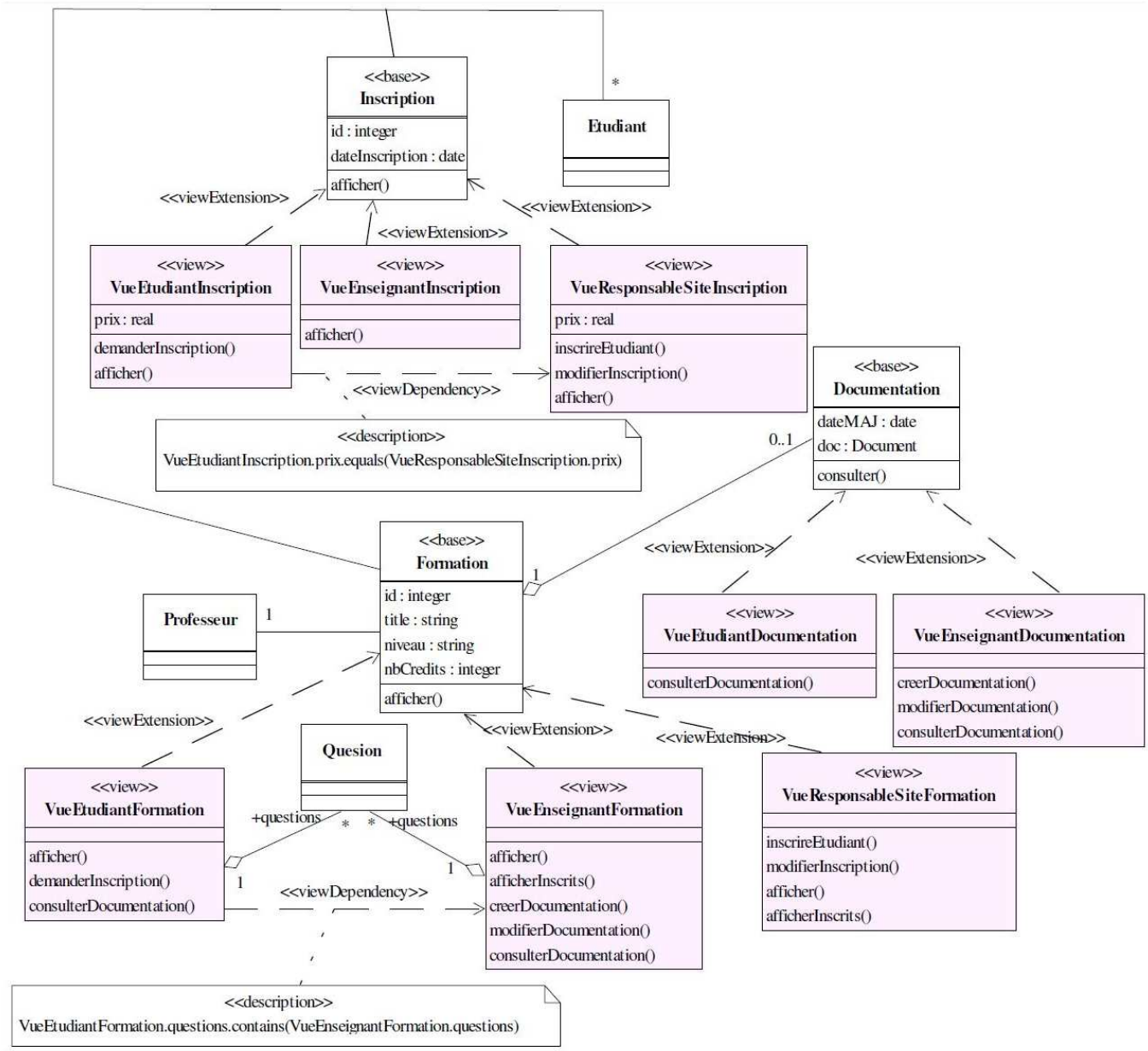


FIGURE 2.9 – Exemple d'un diagramme de classe multivues, extrait de [Nassar et al., 2004].

la vue active. La notion de composants multivues dans l'approche *VUML* a été raffinée et détaillée dans [Nassar et al., 2005, El Asri et al., 2006].
A ses débuts, l'approche *VUML* a adopté une méthodologie de création de diagrammes de classes indépendants pour chaque point de vue, avant de les fusionner manuellement dans un seul diagramme multivues. L'approche tente de rendre le processus de fusion, ou de transformation, semi-automatisé [Nassar et al., 2009, Anwar et al., 2010, Anwar et al., 2011] à travers des règles implémentées en *ATL*.

2.4 Les vues en programmation

La programmation, l'implémentation, ou le codage dans le domaine informatique est l'ensemble des activités qui permettent l'écriture des programmes ou des systèmes informatiques. C'est une phase fondamentale du processus de développement logiciel. Cette phase consiste à la transformation des modèles construits dans la phase de modélisation en code source exécutable implémentant les fonctionnalités et les exigences définies dans ces modèles. Normalement, pour écrire le résultat de cette phase, on utilise un langage de

programmation comme les langages *Java*, *C#*, *C++*, etc.
Dans l'histoire des langages de programmation, les termes décomposition et modularité ont été toujours les mots-clés pour réduire la complexité des programmes. On obtient ainsi des modules moins complexes, faiblement couplés et plus facilement réutilisables.
La séparation des préoccupations, plus connue sous le terme *Separation of concerns* vise à effectuer une modularisation des programmes basée sur des préoccupations (*concerns*) diverses. L'idée est de considérer un système comme un noyau et des extensions. Le noyau est l'ensemble des besoins fonctionnels de base pour lesquels le système est essentiellement conçu et les extensions sont des besoins secondaires, supplémentaires qui ajoutent des fonctionnalités ou des aspects non-fonctionnels au noyau. Cela afin d'aboutir à une implémentation d'un changement statique ou dynamique du comportement des entités logicielles, comme les objets en programmation orientée objet (en anglais *Object-Oriented Programming*, *OOP*) [Rumbaugh et al., 1991] ou les composants en programmation à base de composants (en anglais *Component-based Software Engineering*, *CBSE*) [Oussalah and al., 2005], selon le changement des paramètres de qualité du service, le changement des fonctionnalités offertes par cette entité, ou le changement des droits d'accès et des privilèges à cette entité selon le cas et le statut du processus d'exécution.
Pour atteindre ces objectifs, le développeur a toujours besoin de certaines approches ou langages de programmation lui permettant l'implémentation du noyau du programme à part, et les extensions à part, à travers des attributs et des méthodes supplémentaires responsables d'étendre le comportement de ces entités logicielles.
Plusieurs approches sont proposées pour résoudre ce type de problèmes, surtout en programmation orientée objet, qui soutiennent le concept de point de vue. Ces approches sont parfois appelées techniques de développement orientées aspects, ou *Aspect-oriented development techniques*, parmi lesquelles on peut citer la programmation orientée sujet (*Subjet-oriented programming*), la programmation orientée aspect (*Aspect-oriented programimng*), la programmation orientée vue (*View-oriented programming*), la programmation orientée role (*Role-oriented programming*), et la programmation orientée contexte (*Context-oriented programming*). En plus, des travaux sont munis pour résoudre les problèmes de consistance qui peuvent avoir lieu en appliquant ces techniques de développement orientées aspects.

2.4.1 La programmation orientée sujet

Le paradigme de programmation orienté sujet, ou en anglais *Subject-oriented programming*, est un paradigme qui réfère au paradigme de développement orienté objet en ajoutant quelques particularités. Par ailleurs, dans ce paradigme le statut (attributs) et le comportement (méthodes) des objets ne sont pas définis dans l'objet lui-même à l'instar du paradigme orienté objet, mais ils sont fournis à travers plusieurs perceptions subjectives (*i.e.* Sujet ou *Subject*) associées à l'objet, d'où le nom programmation orientée sujet [Ossher et al., 1995, Harrison and Ossher, 1993].
Le principe de la programmation orientée sujet est d'encourager l'organisation des classes définissant les objets dans plusieurs sujets (*subjects*) qui forment ensemble des sujets plus complexes. Alors, les classes du paradigme orienté objet sont divisées en des fragments de classes dans le paradigme orienté sujet, associés aux différents sujets considérés. Ainsi, chaque sujet peut être vu comme un sous-système orienté objet. Ensuite, des règles de composition doivent être implémentées pour définir comment le système final orienté sujet doit être combiné. Ici, le développeur doit spécifier pour chaque fragment de classe d'un sujet particulier, ses associations avec les fragments de classe des autres sujets, et comment les méthodes de ces fragments de classe vont être combinées.

2.4.2 La programmation orientée aspect

La programmation orientée aspect, ou en anglais *Aspect-oriented programming*, est un paradigme de programmation indépendant des langages de programmation et des technologies particulières, mais qui est souvent appliqué à des langages orientés objet comme le langage *Java* dans le projet *AspectJ* [Laddad, 2009]

de la plateforme *eclipse*.
Ce paradigme de programmation permet d'implémenter séparément des préoccupations transversales (*Cross-cutting concerns*) qui révèlent souvent des préoccupations non-fonctionnelles ou de la technique (en anglais *Aspect*) qui coupe transversalement les besoins fonctionnels du système, ou le noyau de ce système. D'où le nom programmation orientée aspect [Mcheick et al., 2006, Mili et al., 1999, Kiczales et al., 1997].
Le principe de la programmation orientée aspect est de séparer les modules techniques des modules métiers. Ainsi, au lieu de faire appel aux modules techniques depuis les modules métiers, le développeur se concentre en premier lieu sur le logique métier de son système. Ensuite, les aspects techniques seront spécifiés d'une façon autonome, en écrivant leur code séparément, et en attribuant à chaque module technique (aspect) un ensemble de points d'insertion (en anglais *joint-point*) pour établir la liaison entre ces aspects et les modules métiers ou techniques impliqués. Finalement, la fusion du code technique, ou des aspects avec le code métier sera réalisée, soit à la compilation où à l'exécution par un tisseur d'aspect (en anglais *Weaver*) spécifique au langage de programmation.
Même si le paradigme de développement orienté aspect fait partie de la phase implémentation dans le processus de développement d'un logiciel, plusieurs approches appliquent la notion orientée aspect dans la phase de spécification de besoins, comme dans [Majumdar and Bhattacharya, 2010].

2.4.3 La programmation orientée vue

La programmation orientée vue décrite dans [Mili et al., 1999] et l'extension de la programmation orientée objet pour intégrer la notion de vue introduite dans [Shiling and Sweeney, 1989] adopte des principes très proches aux principes de la programmation orientée sujet, et se basent sur la définition d'interfaces multiples pour chaque objet qui sont réellement associées à plusieurs vues.
Ainsi, ce paradigme de développement se base sur l'implémentation d'un noyau du système informatique en écrivant le code des objets, et l'implémentation des extensions sous la forme de vues du système représentées par des interfaces. Ces vues permettent aux programmes clients d'accéder plusieurs champs fonctionnels (ou vues) simultanément, et leur permettent aussi d'ajouter ou de supprimer des morceaux fonctionnels (*functional slices*) pendant le temps d'exécution.
En effet, la combinaison de ces différents champs fonctionnels ou vues associées à un objet se fait en créant une couche autour de l'objet et ses vues, dénotée "*Wrapper*", responsable de recevoir tous les appels des méthodes de cet objet, et les rediriger vers le composant approprié en se basant sur le nombre et le statut (active ou inactive) des vues attachées à l'objet.
L'importance de ce paradigme se manifeste dans le fait que les développeurs n'ont pas besoin de maîtriser de nouveaux langages afin d'appliquer cette approche, mais au contraire, ils peuvent juste utiliser une interface de programmation (*Application Programming Interface, API*) spécifique fournie pour manipuler les vues d'un objet durant l'exécution (addition, suppression, activation, désactivation).

2.4.4 La programmation orientée rôle

Le paradigme de programmation orienté rôle [Graversen, 2006], ou *Role-oriented programming*, est un paradigme qui se base sur l'expression des systèmes informatiques et de leurs composants d'une manière analogue à la conception et la compréhension humaine du monde, afin de rendre plus facile aux développeurs des logiciels la compréhension et la maintenance de ces systèmes.
Alors, un objet dans le paradigme orienté rôle, peut avoir une définition et un comportement intrinsèque à lui, commun pour n'importe quel rôle qu'il peut prendre. De plus, il peut avoir plusieurs autres définitions et comportements pour chacun des rôles qu'il peut prendre, définis à travers des interfaces associées à l'objet. Normalement, un objet peut changer de rôles dynamiquement pendant le temps d'exécution du système, ce qui permet de mettre en oeuvre la notion d'évolutivité des objets.
En effet, le paradigme de programmation orienté rôle ressemble beaucoup au paradigme orienté vue, puisque ces deux paradigmes se basent sur la définition d'un objet et de plusieurs interfaces associées à

cet objet, représentant respectivement les rôles et les vues considérées pour cette objet.

2.4.5 La programmation orientée contexte

Le paradigme de programmation orienté contexte, ou *Context-oriented programming* (*COP*), présenté dans [Hirschfeld et al., 2008], est un paradigme de programmation considéré successeur au paradigme de programmation orienté sujet. Ce paradigme adresse le problème de changement dynamique du comportement du système informatique en se basant sur les évolutions occurrentes vis-à-vis du contexte ou les conditions dans lesquels le système s'exécute. En effet, trois facteurs fondamentaux définissent le contexte de l'exécution : (1) l'acteur ; (2) l'environnement ; (3) le système.
Le principe de ce paradigme se base sur l'expédition, ou en anglais *Dispatching*, des messages envoyés d'une manière multidimensionnelle. Ainsi, le message sera expédié ou redirigé au bout de code approprié selon quatre paramètres. Le premier paramètre est l'acteur envoyant le message, le deuxième est l'objet qui reçoit le message, le troisième est la méthode demandée, et finalement le quatrième paramètre est le contexte dans lequel le message est envoyé. Notamment, le premier paramètre ou la première dimension est appliquée dans la programmation impérative, le deuxième et le troisième sont appliqués en orienté objet, les trois premiers en orienté sujet, et les quatre paramètres représentent la contribution du paradigme de programmation orienté contexte.

2.4.6 Les travaux sur la consistance entre les vues

Autre que les paradigmes de développement et d'implémentation adoptant explicitement, ou implicitement la notion multipoints de vue, on peut trouver des approches et des travaux qui ont élaboré le problème de la consistance et la cohérence dans les systèmes multipoints de vue au niveau de l'implémentation. Et cela afin de suivre la trace d'un changement ou d'une modification qui a eu lieu quelque part dans une vue particulière, et détecter les changements et les modifications candidates qui doivent avoir lieu afin de préserver la cohérence du code source. Par exemple, dans [Grundy et al., 1998], un outil de support est développé pour : (1) détecter tous les changements qui doivent avoir lieu suite à un changement particulier au niveau du nom d'une classe, ses méthodes, ou ses attributs ; (2) ensuite envoyer des notifications aux développeurs concernés pour effectuer les modifications nécessaires.

2.5 Les vues en architecture logicielle

En architecture logicielle, il est quasiment impossible de capturer l'architecture d'un système complexe dans un seul modèle qui soit compréhensible par tous les intervenants. Par intervenant, plus connu sous le nom de *stakeholder*, on englobe aussi bien l'utilisateur du futur système que le constructeur de ce dernier. Un constructeur est une personne ou un groupe de personnes qui conçoit, développe, teste, déploie ou répare le futur système.
Chacune de ces activités implique que chaque intervenant a ses propres besoins, intérêts et exigences et souhaite que le système en tienne compte. La compréhension du rôle et les aspirations de tout intervenant est le rôle de l'architecte lors du développement du système.
La solution préconisée pour la description d'une architecture (*Architectural Description*) est de la partitionner en un nombre de vues séparées et interdépendantes, qui collectivement décrivent les aspects fonctionnels et non-fonctionnels (telle que la performance, robustesse, la disponibilité, la concurrence, la distribution, etc.) du système. Cette solution n'est pas nouvelle ; en effet, elle remonte aux travaux de *Parnas* [Parnas, 1971] et de *Perry et Wolf* [Perry and Wolf, 1992].
Plusieurs études ont été menées sur les approches d'architectures logicielles dans la littérature, comme le travail présenté dans le livre "*Documenting Software Architectures : views and beyond*" [Clements, 2003] par *P. Clements et al.*, dans lequel les auteurs définissent leur modèle de vue et le comparent ultérieurement avec trois autres modèles qui sont le standard *IEEE 1471* (le prédécesseur du standard *IEEE 42010*), *UML*,

et la plateforme *C4ISR*. Egalement, une étude a été munie dans le livre "*Architectures logicielles : Principes, techniques et outils*" [Oussalah and al., 2014] par *M. Oussalah et al.*, dans lequel les auteurs ont aussi proposé leur propre modèle.
Une autre étude a été effectuée dans [Smolander, 2002] par *K. Smolander*, sur les pratiques de documentation des architectures logicielles. Cette étude a couvert plusieurs approches basées sur les points de vue comme le standard *IEEE 1471*, *RM-ODP*, *"4+1" View Model*, la plateforme *Zachman*, et d'autres approches.
Par ailleurs, une étude importante et fondamentale a été effectuée par *Nicholas May* en 2005 [May, 2005], dans laquelle une comparaison est effectuée, basée sur une plateforme de comparaison représentée par le standard *IEEE 1471*, entre cinq approches d'architectures logicielles multipoints de vue qui sont le modèle *"4+1" View Model*, le modèle *Views and Beyond*, *RM-ODP*, le modèle *Siemens*, et l'approche *ADS*. Enfin, en se basant sur l'étude comparative, un ensemble de points de vue optimal et couvrant l'espace le plus important des systèmes informatiques est déduit.
Dans ce qui suit, nous détaillons huit approches ayant apporté des solutions satisfaisantes à cette problématique. Nous citons :

- le modèle *"4+1" View Model* de *P. Kruchten* en 1995, présenté dans le paragraphe 2.5.1 ;
- l'approche *Rational Architectural Description Specification*, ou *Rational ADS*, qui est développée en 2004 comme une expansion du *"4+1" View model*. Cette approche sera présentée dans le paragraphe 2.5.2 ;
- le standard *ISO/IEC/IEEE 42010* de *IEEE Architecture Planning Group* (*APG*) qui fut approuvé en 2000 par le *IEEE-SA standards board* et codé *IEEE 1471-2000*. En 2007, ce standard fut adopté par le *ISO/IEC JTC1/SC7* et nommé *ISO/IEC 42010 :2007* avant qu'il soit publié en 2011 sous le nom *ISO/IEC/IEEE 42010*. Ce standard sera présenté dans le paragraphe 2.5.3 ;
- l'approche *Views and beyond* (*V&B*) pour la documentation des architectures logicielles, développée par le *SEI* (*Carnegie Mellon Software Engineering Institute*), cette approche sera présentée dans le paragraphe 2.5.4 ;
- l'approche de *Rozanski et Woods* développée en 2011 conforme au standard IEEE 42010. Cette approche sera présentée dans le paragraphe 2.5.5 ;
- le modèle d'architecture logicielle de *Siemens* proposé en 1992 par *D. Soni et al.*, ce modèle sera présenté dans le paragraphe 2.5.6 ;
- la plateforme de *Zachman* qui a été conçue en 1987 afin d'organiser les modèles d'une architecture logicielle, et évoluée jusqu'à 2008 quand elle est devenue une plateforme d'architectures d'entreprises. Cette plateforme sera présentée dans le paragraphe 2.5.7 ;
- la plateforme *RM-ODP* (*ISO Reference Model of Open Distributed Processing*) développée en 1995 comme étant une plateforme d'architectures logicielles associée à la modélisation des systèmes informatiques distribués. Cette plateforme sera présentée dans le paragraphe 2.5.8.

2.5.1 Le modèle *"4+1" View Model*

Le modèle "4+1" [Kruchten, 1995] a été proposé par *P. Kruchten* du *Rational Software Corp.* Ce modèle repose sur cinq vues principales, illustrées dans la figure 2.10 qui sont :

- la vue logique (*Logical view*) recouvre principalement les besoins fonctionnels du système, ou en d'autres termes ce que le système fournit comme services aux utilisateurs. La vue logique représente les objets du système et les relations qui existent entre eux ;

- la vue processus (*Process view*) représente un aspect non-fonctionnel des besoins du système comme la performance, la disponibilité, la concurrence, la distribution, l'intégrité, etc. Cette vue divise le système en un ensemble de processus et représente les interactions qui auront lieu entre ces processus ;
- la vue de développement (*Development view*) se concentre sur l'organisation modulaire du système. Ainsi, dans cette vue le système sera décomposé en plusieurs librairies ou sous-systèmes qui peuvent être implémentés chacun par un développeur ou une mini-équipe de développeurs. Cette vue recouvre les besoins internes de l'implémentation comme la réutilisabilité, le respect des standards de l'implémentation, le respect des contraintes imposées par les outils utilisés, etc. ;
- la vue physique (*Physical view*) prend en compte principalement des besoins non-fonctionnels du système comme la disponibilité, la fiabilité, la performance, l'évolutivité, etc. Dans la vue physique, les éléments logiciels comme les processus et les objets seront associés aux différents éléments physiques comme les processeurs et les disques durs ;
- les scénarios (*Scenarios*), représentent l'intégration des éléments des quatre vues précédentes en utilisant un petit ensemble de scénarios qui paraissent importants.

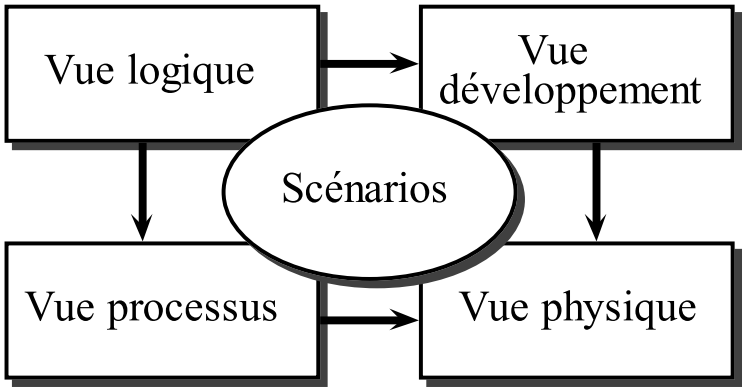


FIGURE 2.10 – Le modèle "4+1" de *Kruchten*, extrait de [Kruchten, 1995].

Les vues proposées dans ce modèle ne sont pas complétement indépendantes. En effet, des correspondances peuvent être définies entre ces vues, comme par exemple la correspondance pratique entre une classe ou un petit ensemble de classes qui sera normalement représenté sous forme d'un module ou un ensemble de modules dans la vue de développement, ou le processus de la vue processus qui sera associé à des supports physiques de la vue physique sur lesquels il doit être exécuté.
Il est à noter que ce modèle a été adopté dans le processus de développement itératif *RUP* (*Rational Unified process*).

2.5.2 La spécification rationnelle de la description d'architecture (*ADS*)

L'approche *Rational ADS* (*Rational Architecture Description Specification*) [Norris, 2004] est une expansion du *"4+1" View Model* de *P. Kruchten*. Elle a été conçue afin d'avoir un meilleur comportement et support avec des systèmes plus complexes qu'ils soient informatiques ou non. Selon l'étude de *Nicholas May* [May, 2005], cette approche dispose d'une définition formelle de l'évolution des besoins et de la testabilité des architectures.
En effet, la plupart des vues du modèle de *Kruchten* ont été renommées, et de nouvelles autres vues ont été ajoutées et organisées dans quatre points de vue distincts. Ainsi, la vue "*Scenarios*" a été renommée et devenu "*Use Case View*", et a formé le noyau du premier point de vue "*Requirements Viewpoint*" dans lequel ils ont ajouté trois autres vues, qui sont "*Non Functional Requirements View*", "*Domain View*", et "*User Experience View*". Ensuite, les deux vues "*Logical View*" et "*Process View*" ont construit le point de

vue "*Design Viewpoint*", et les deux autres vues "*Development View*" et "*Physical View*" ont été renommées pour qu'elles seront respectivement "*Implementation View*" et "*Deployment View*", et ils ont formé le point de vue "*Realization Viewpoint*". Finalement, le quatrième point de vue a regroupé une nouvelle vue unique qui est "*Test View*".
Dans cette approche, la consistance de l'architecture est effectuée à travers plusieurs relations fixes et explicites définies entre les différentes vues, représentées dans la figure 2.11.

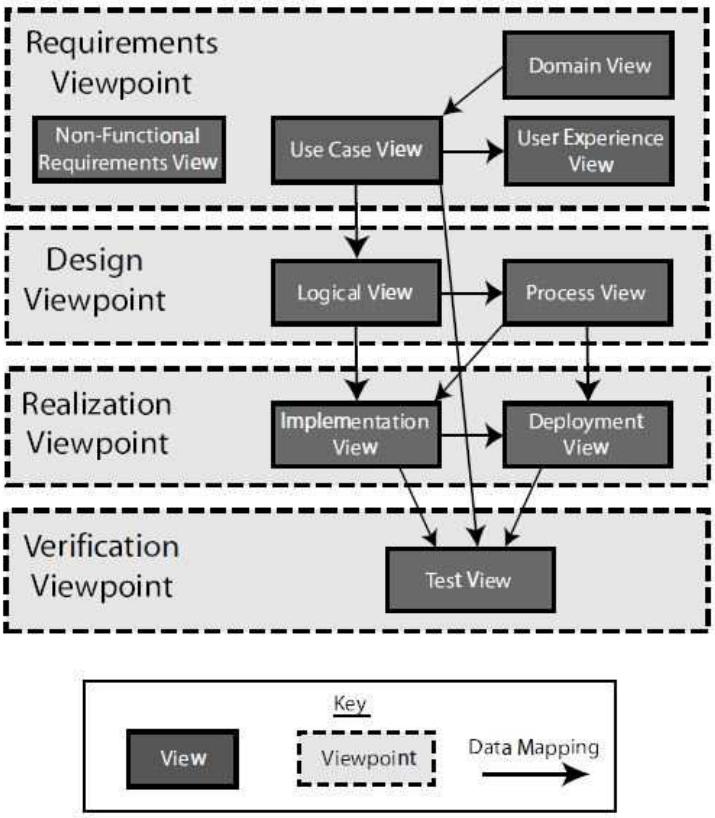


FIGURE 2.11 – Le modèle *ADS*, extraite de [May, 2005].

2.5.3 Le standard *ISO/IEC/IEEE 42010*

Le standard *ISO/IEC/IEEE 42010* [ISO/IEC/IEEE, 2011] a été conçu par le *APG* (*The IEEE Architecture Planning Group*) afin de formaliser la définition d'une architecture logicielle et ses éléments principaux, et aussi dans le but de fournir un standard commun visant l'incorporation et l'incarnation des efforts effectués dans ce domaine. Ce standard définit une architecture comme étant l'organisation d'un système, structurée par une collection de composants (c'est-à-dire unités) logiciels et des liens ou relations définies entre ces composants.
La figure 2.12 illustre le modèle proposé dans le standard *IEEE 42010*. Selon ce modèle :

- un point de vue (*Viewpoint*) est une spécification des conventions de construction des vues qui lui appartiennent. Ces conventions peuvent être définies dans l'architecture elle-même ou importées d'une entité externe appelée *Model kind* ;
- une vue (*View*) est conforme à un point de vue et consiste en un ensemble de modèles ;
- une description d'architecture s'adresse à un ensemble d'intervenants (*stakeholder*) qui ont des intérêts ou des préoccupations (*concern*). Un *concern* peut être couvert par des *Viewpoints*. Aussi, une description d'architecture est composée d'un ensemble de vues.

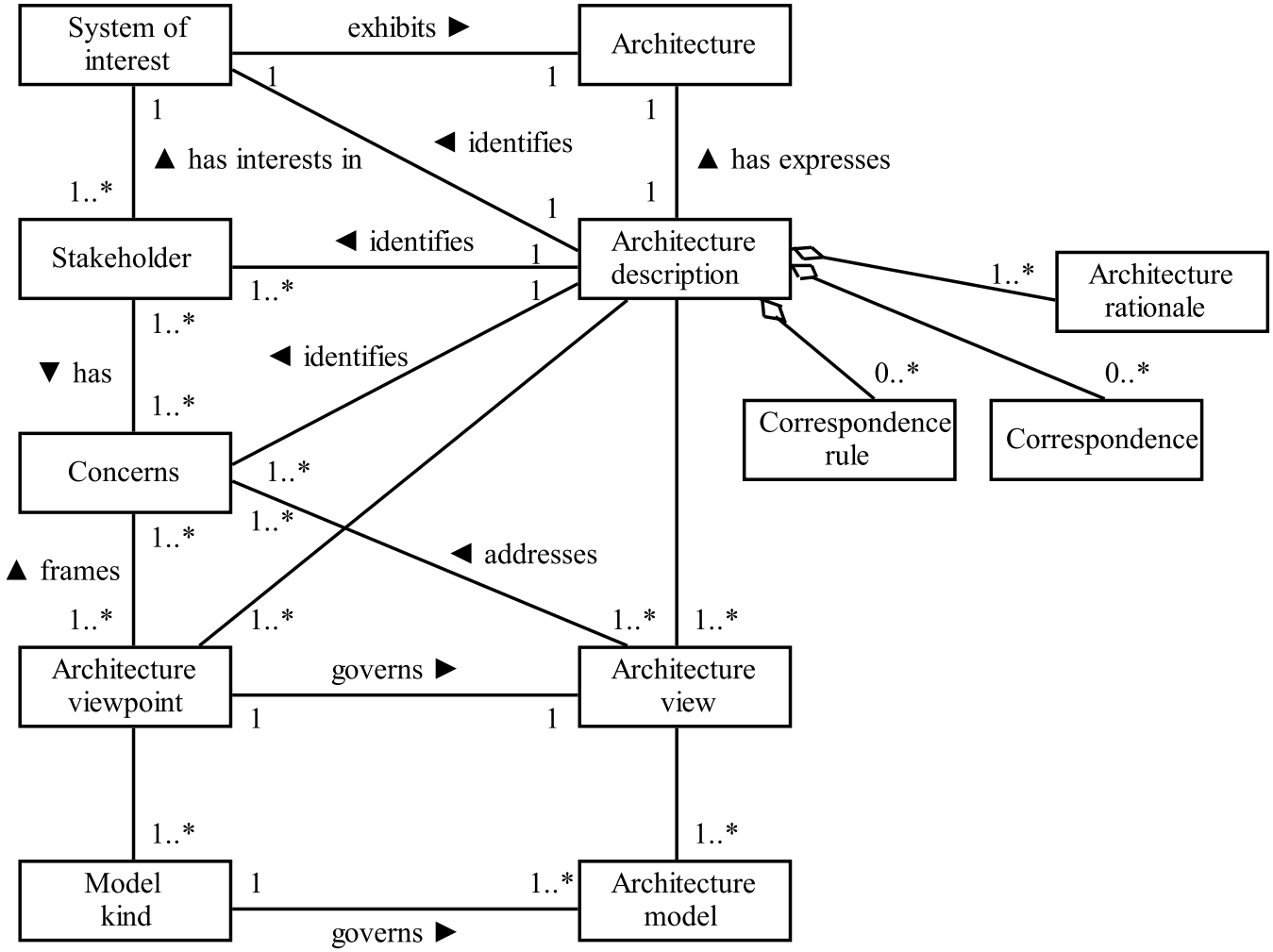


FIGURE 2.12 – Le modèle proposé dans le standard *IEEE 42010*, extrait de [ISO/IEC/IEEE, 2011].

- Une correspondance (*Correspondence*) définit une relation entre deux éléments architecturaux, qu'ils soient des intervenants, préoccupations, points de vue, vues, ou n'importe quels éléments de l'architecture. Alors, ces correspondances sont utilisées normalement pour exprimer les relations qui peuvent avoir lieu, au sein d'une description d'architecture, entre les intérêts apportés par les éléments architecturaux impliqués.

2.5.4 L'approche *Views and beyond* (*V&B*)

P. Clements et al. ont développé au sein du *SEI* (*Carnegie Mellon Software Engineering Institute*) l'approche *Views and beyond* (*V&B*) [Clements et al., 2002] pour la documentation des architectures logicielles. Cette approche comme son nom l'indique, utilise les vues pour aboutir à une organisation fondamentale des architectures logicielles. En effet, *V&B* se base sur le principe que la documentation d'une architecture logicielle commence d'abord par la documentation des vues pertinentes de cette architecture et ensuite, en documentant les informations liant les vues entre elles.
V&B s'adresse, comme dans le cas du standard *IEEE 42010*, à tous les intervenants durant le processus de développement du système informatique, et vise à fournir une documentation d'une architecture logicielle qui soit déclinée en plusieurs vues s'adressant aux différents intervenants et répondant à leurs besoins.
Dans cette approche, une hiérarchie à trois niveaux est définie :

- les *Viewtypes* : à l'instar des *Viewpoints* définis dans le standard *IEEE 42010*, un *Viewtype* représente une catégorie de vues et s'adresse à un ou un ensemble d'intervenants. Il existe trois *Viewtypes* :

  - *Module Viewtype* représente la structure du système en termes d'un ensemble d'unités de code,
  - *Component & Connector Viewtype* (*C&C*) représente un ensemble d'éléments interagissant durant l'exécution,
  - *Allocation Viewtype* représente la relation entre les éléments logiciels et les éléments non-logiciels comme les machines et les processeurs ;

- les *Styles* : un style architectural aussi nommé pattern architectural représente un pattern de haut niveau qui aide à spécifier la structure fondamentale d'une application. Tout style aide à réaliser une propriété globale du système, telles que l'adaptabilité de l'interface utilisateur ou la distribution. Les styles sont regroupés dans les *Viewtypes* qui sont considérés comme des catégories de styles. Par exemple, le modèle client-serveur représente un style pour le *viewtype C&C*. La liste des styles définie pour chaque *viewtype* est illustrée dans la figure 2.13. Il est à noter que les notions de style et de catégorie de style ne sont pas propres à *V&B* mais déjà proposées et largement commentées dans d'autres travaux en architecture logicielle, notamment dans le livre de *Bushmann et al.* [Buschmann et al., 1996] ;

- les *Views* : représentent des collections d'éléments du système et les relations qui les associent. Notons que les vues d'une architecture sont documentées selon un template défini par les concepteurs de cette approche. En effet, une vue d'une architecture est toujours conforme à un style d'un *viewtype*.

Viewtypes	Styles	Vues
Module	Decomposition	Les styles appliqués à des systèmes particuliers
	Généralisation	
	Utilisation	
	En couche	
Composant et connecteur	Pipe-and-filter	
	Données partagées	
	Communicating-processes	
	Peer-to-peer	
	Client-serveur	
Allocation	Affectation de travail	
	Déploiement	
	Implémentation	

FIGURE 2.13 – L'ensemble des viewtypes, styles et vues dans *V&B*, extrait de [Clements et al., 2002].

V&B propose un guide de trois étapes pour la sélection des vues pertinentes nécessaires pour documenter l'architecture d'un système :

1. produire une liste de vues candidates : cette étape consiste à construire un tableau à deux dimensions (ligne-colonne) telle que les lignes représentent les intervenants (*stakeholders*) que l'architecte juge pertinents pour le projet en cours et les colonnes dénotant les vues regroupées en *Viewtypes* et les styles qui peuvent être appliqués. Ensuite, l'architecte procède à remplir les cases du tableau par des valeurs décrivant le niveau d'information requis pour chaque *stakeholder* et style. Le niveau d'information peut être : *d* (*detailed information*) pour information détaillée, *s* (*Some detail*) pour peu de détails et *o* (*Overview Information*) pour information générale ;

2. combiner des vues : cette étape vise à minimiser le nombre des vues obtenu en ignorant les vues dont la valeur architecturale est couverte dans d'autres vues et en combinant d'autres vues. Combiner des vues consiste à produire une vue appelée vue combinée à partir d'autres vues ; cela revient à combiner des éléments provenant de deux ou plusieurs vues en un seul. Par exemple, dans de petits ou moyens projets, les vues *Work Assignment* et *Implementation* peuvent se superposer avec la vue *module decomposition* ;

3. trier les vues : le tri des vues restantes s'effectue selon l'ordre de priorité de documentation, en fonction des détails spécifiques du projet.

La dernière phase de la documentation d'une architecture proposée par l'approche *V&B* est la documentation des informations inter-vues qui sont appliquées à plusieurs vues. Cette phase a été proposée afin de relier les vues de l'architecture entre elles et de donner une image globale de l'architecture facilitant sa compréhension par les *stakeholders*. En effet, cette phase peut être divisée selon les trois étapes suivantes :

- la spécification de l'organisation globale en fournissant un plan de la documentation toute entière ;
- la description de l'architecture en donnant un panorama du système, un index de tous les éléments de l'architecture, un glossaire et une liste des acronymes ;
- la justification de l'architecture finale et des décisions prises pour aboutir à cette architecture.

2.5.5 L'approche de Rozanski et Woods

L'approche présentée dans [Rozanski and Woods, 2011], est conforme avec le standard *IEEE 42010* présenté auparavant. Dans cette approche, une architecture logicielle est décomposée en plusieurs vues représentant chacune des aspects structurels de cette architecture et un ensemble de préoccupations des intervenants impliqués dans l'architecture. Une vue doit obéir à un ensemble de partons et de conventions qui vont être utilisées dans la construction de ses artefacts ; ces patrons et conventions sont regroupées dans des points de vue.
Cette approche fournit un catalogue de sept points de vue différents qui sont :

- Contexte (*Context*) : décrit les relations, dépendances, et interactions entre le système et son environnement ;
- Fonctionnel (*Functional*) : décrit les éléments fonctionnels du système pendant l'exécution, leurs responsabilités, interfaces, et interactions ;
- Information : décrit comment le système sauvegarde, manipule, gère, et distribue les informations et les données ;
- Concurrence (*Concurrency*) : décrit la structure de la concurrence du système et attribut des éléments fonctionnels à des unités de concurrence pour clarifier les portions du processus d'exécution du système là où on peut avoir des opérations concurrentes ;

- Développement (*Development*) : décrit et communique les aspects architecturaux avec les intervenants concernés dans la construction du système afin d'avoir un processus de développement convenable ;
- Déploiement (*Deployment*) : décrit l'environnement dans lequel le système va être déployé, et les dépendances potentielles que le système peut avoir entre ces éléments ;
- Opérationnel (*Operational*) : décrit comment le système va être manipulé, géré, et soutenu durant son exécution dans un environnement de production.

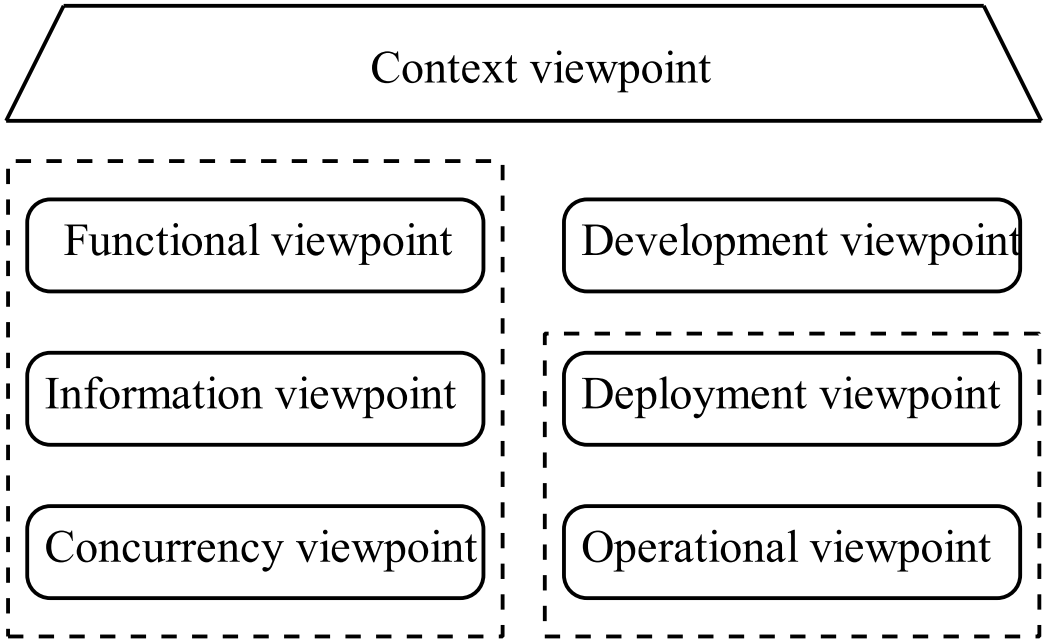


FIGURE 2.14 – Le groupement des points de vue dans l'approche de *Rozanski et Woods*, extrait de [Rozanski and Woods, 2011].

Ces points de vue constituent un guide pour l'architecte lors de la définition de l'architecture logicielle. Ainsi, selon le contexte, il sera amené à choisir les points de vue qu'il juge pertinents pour faire communiquer l'architecture à l'ensemble des intervenants. Par contexte, on entend l'étape courante durant le cycle de développement du logiciel ainsi que le type et les attentes des intervenants à un moment donné.
Dans ce catalogue, présenté dans le diagramme de la figure 2.14,

- le point de vue *Contexte* est placé en haut du diagramme pour indiquer son rôle global dans la définition du cadre et du contenu des autres points de vue.
- Les points de vue *Fonctionnel*, *Information*, et *Concurrence* détaillent comment le système va réaliser les fonctionnalités définies auparavant ; Ils sont donc regroupés à gauche du diagramme.
- Le point de vue développement, définit la manière à avoir les fonctionnalités du système.
- Les point de vues déploiement et opérationnel, indiquent comment le système développé auparavant va être déployé et maintenu. Ils sont regroupés car ils représentent ensemble, le système dans son environnement de production.

La consistance et la cohérence entre les différentes vues de l'architecture sont assurées à travers un ensemble de relations entre ces différentes vues. Dans la figure 2.15, ces relations sont représentées par des flèches. Une flèche exprime une dépendance entre deux vues : un changement dans la vue d'arrivée exige en général

des changements dans la vue de départ. De plus, un ensemble de recommandations et de points de contrôle, prodigués à l'issue de la construction d'un modèle selon un point de vue, exprimés sous forme de questions auxquelles l'architecte est amené à répondre, afin de vérifier la cohérence de l'architecture.
Aussi, dans cette approche le concept de *Perspective* est introduit. Une perspective est définie comme une

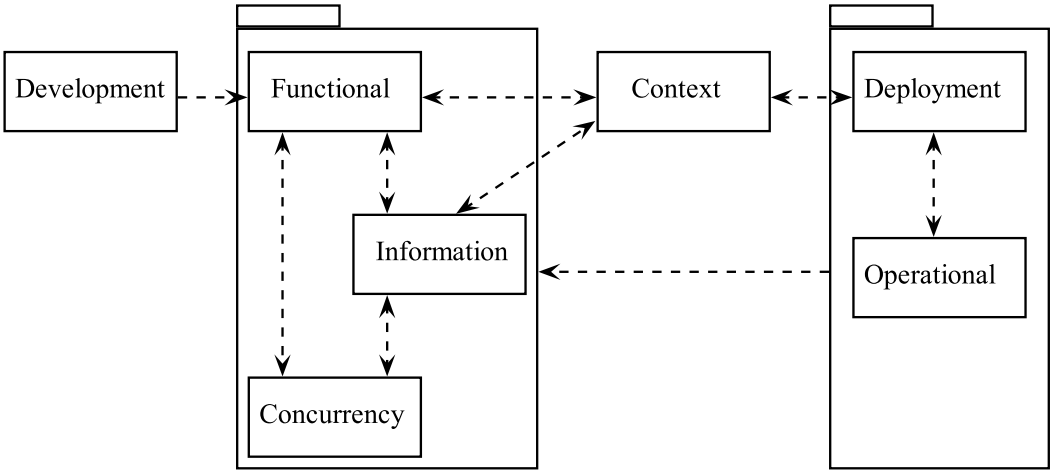


FIGURE 2.15 – Les relations entre les points de vue dans l'approche de *Rozanski et Woods*, extraite de [Rozanski and Woods, 2011].

collection d'activités et de tactiques utilisés pour assurer que le système respecte des propriétés qualitatives de l'architecture du système. Parmi les perspectives qui sont définies dans le livre de *Rozanski et Woods* [Rozanski and Woods, 2011] on peut citer : *Sécurité*, *Performance*, *Evolutivité*, *Disponibilité*, *Accessibilité*, Etc. ...
Finalement, pour pouvoir appliquer les concepts définis dans cette approche, un processus de définition d'architecture logicielle a été proposé. Ce processus est représenté dans un diagramme d'activité.

2.5.6 Le modèle de Siemens

Selon l'étude de [Soni et al., 1995], les architectures logicielles sont regroupées dans quatre catégories :

(1) les architectures conceptuelles ;

(2) les architectures de modules ;

(3) les architectures d'exécution ;

(4) les architectures de code source.

Ensuite, les auteurs ont prouvé qu'aucune parmi ces architectures ne peut être considérée suffisante pour documenter une architecture logicielle, et que selon le domaine d'application des systèmes informatiques, c'est toujours indispensable de faire la combinaison de plusieurs architectures parmi les quatre citées.
Par la suite, un modèle d'architecture logicielle est proposé, ayant quatre vues distinctes associées aux quatre catégories de l'étude. En plus, des relations entre ces quatre vues ont été définies ; alors que les éléments de la vue conceptuelle sont implémentés dans des modules de la vue *Module*, et associés à des éléments de le vue *Execution*. D'autre part, les éléments de la vue des *Module* sont implémentés par des éléments de la vue *Code*, et les éléments de la vue *Code* peuvent configurer des éléments *Execution*.
La figure 2.16 illustre les différentes vues du modèle *Siemens* et les relations existantes entre elles.

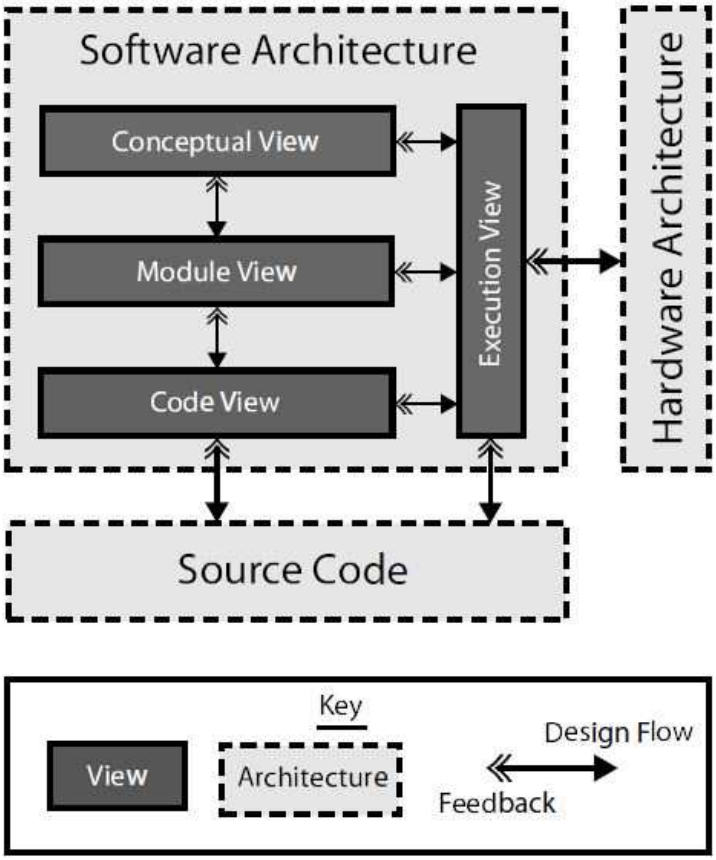


FIGURE 2.16 – Le modèle *Siemens* de quatre vues, extraite de [May, 2005].

2.5.7 La plateforme de *Zachman*

La plateforme de *Zachman* [Zachman, 1987] est conçue afin d'organiser et de classifier les modèles et les artefacts d'une architecture logicielle. Cette plateforme a été étendue en 1992 dans [Sowa and Zachman, 1992], et en 1997 dans [Zachman, 1996]. En 2008, elle fut présentée sous la forme d'un nouveau standard [Technology, 2008]. Ainsi, une architecture logicielle est considérée à travers une matrice bidimensionnelle 6 X 6, ayant six perspectives dans les lignes et six transformations dans les colonnes. Les perspectives sont au nombre de six et sont détaillées dans la table 2.1.
Les colonnes de la matrice représentent l'intérêt, ou le *focus*, des artefacts et des modèles répondant à six questions fondamentales pour chaque perspective. Les questions (ou focus) sont :

- Quoi (*What*) - la description des données,
- Comment (*How*) - la description des fonctions,
- Où (*Where*) - la description du réseau,
- Qui (*Who*) - la description des acteurs,
- Quand (*Where*) - la description du temps,
- Pourquoi (*Why*) - la description des motivations.

La figure 2.17 représente le contenu des modèles et des artefacts associés selon les perspectives et focus. En effet, chaque ligne de la matrice représente une vue complète d'une perspective particulière du système en considération. Par conséquence, les lignes supérieures de la matrice représentant les perspectives supérieures n'ont pas nécessairement un niveau de compréhension globale du système plus élevé que celui

	Why	How	What	Who	Where	When
Contextual	Goal List	Process List	Material List	Organizational Unit & Role List	Geographical Locations List	Event List
Conceptual	Goal Relationship	Process Model	Entity Relationship Model	Organizational Unit & Role Rel. Model	Locations Model	Event Model
Logical	Rules Diagram	Process Diagram	Data Model Diagram	Role relationship Diagram	Locations Diagram	Event Diagram
Physical	Rules Specification	Process Function Specification	Data Entity Specification	Role Specification	Location Specification	Event Specification
Detailed	Rules Details	Process Details	Data Details	Role Details	Location details	Event Details

FIGURE 2.17 – Le contenu des modèles dans la plateforme de *Zachman*.

associé aux lignes inférieures. Pourtant, les artefacts des perspectives supérieures doivent fournir les détails suffisants pour la définition des artefacts des perspectives inférieures. De même, chaque perspective doit tenir en compte des préoccupations des autres perspectives, et des contraintes des perspectives supérieures.

La figure 2.18 illustre les différentes perspectives proposées par *Zachman*.

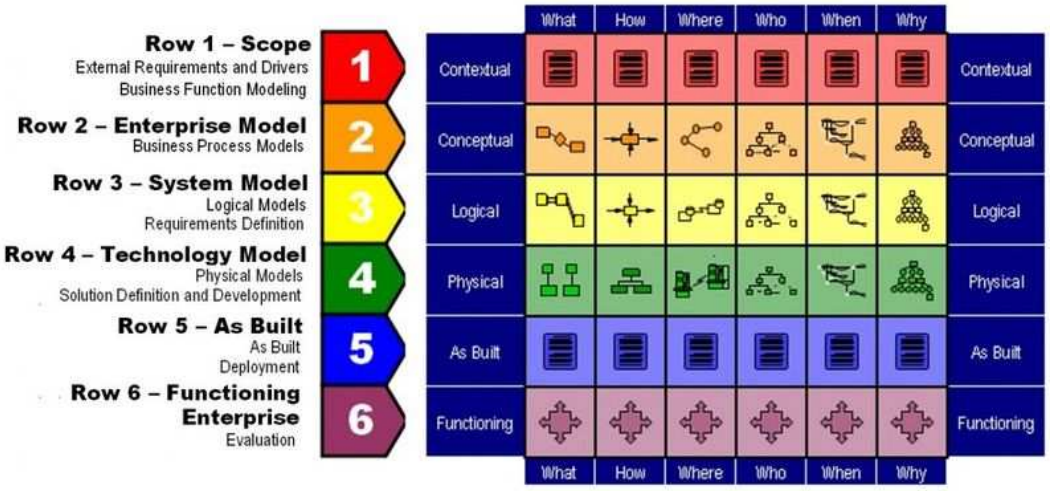


FIGURE 2.18 – La représentation d'une architecture logicielle par une matrice bidirectionnelle.

2.5.8 *RM-ODP*

RM-ODP (*Reference Model of Open Distributed Processing*) [Raymond and Armstrong, 1995, Raymond, 1995], est une plateforme d'architectures logicielles associée à la modélisation des systèmes informatiques complexes et distribués. C'est un modèle de référence en informatique qui offre une plateforme coordinatrice pour la standardisation du *ODP* (*Open Distributed Processing*). Cette approche offre cinq points de vue distincts, génériques, et complémentaires vis-à-vis d'un système informatique et son environnement. Ces points de vue sont illustrés dans la figure 2.19 et sont comme suit :

- le point de vue de l'entreprise (Entreprise viewpoint),

TABLE 2.1 – Les différentes perspectives de la plateforme de *Zachman*

Perspective	Intervenant	Description
Portée (*Scope*)	Planificateur (*Planner*)	C'est une estimation de la portée du système, ses coûts, ses interaction avec l'environnement, etc.
Modèle métier (*Business Model*)	Propriétaire (*Owner*)	C'est le produit final du point de vue du propriétaire ; la conception et les modèles métiers.
Modèle d'information (*Information systems model*)	Concepteur (*Designer*)	C'est la modélisation détaillée du système en se basant sur le modèles métiers.
Modèle de technologie (*Technology Model*)	Constructeur (*Builder*)	C'est l'adaptation de la modélisation du système avec les technologies, outils, et langages de programmation utilisées.
Spécification détaillée (*Detailed Specification*)	*Subcontractor*	Correspond à l'implémentation de chaque module à part.
Vue du système actuel		Correspond au système qui tourne.

- le point de vue de l'information (Information viewpoint),
- le point de vue de l'informatisation (Computational viewpoint),
- le point de vue de l'ingénierie (Engineering viewpoint),
- le point de vue de la technologie (Technology viewpoint).

En 1995, *Bowman et al.* ont commencé à travailler sur la définition d'une consistance entre les différents points de vue *RM-ODP* [Bowman et al., 1995]. Puis en 2002, ils ont abouti à formaliser leur définition d'une consistance entre les différents points de vue du système [Bowman et al., 2002]. Ainsi, une définition formelle de la notion de consistance est proposée. Ici, deux catégories de consistance sont définies : (1) la consistance *intra-langage*, qui peut exister entre deux spécifications exprimées par un seul langage ; (2) la consistance *inter-langage*, qui peut exister entre des spécifications exprimées par plusieurs langages. La définition de consistance et de type de vérification offert dans cette approche prend toujours en compte la nature des relations entre les différents points de vue, de manière que les opérations de vérification de consistance entre, par exemple, le point de vue *Entreprise* et le point de vue *Information* se diffère des opérations de vérification de consistance entre deux autres points de vue, comme le point de vue *Computational* et le point de vue *Technology*. Cela est due au fait que les points de vue dans *RM-ODP* sont toujours fixes. À noter que cette approche prend en considération la consistance globale de l'architecture logicielle, c.à.d. la consistance entre tous les points de vue de l'architecture, non pas juste la consistance binaire, ou la consistance entre deux point de vue.

2.5.9 Les travaux de *Rich Hilliard*

Rich Hilliard a élaboré beaucoup de travaux à propos des architectures logicielles multipoints de vue. Parmi ces travaux, une extension du standard *IEEE 1471* est proposée, dans [Hilliard, 2007], afin d'appliquer sur

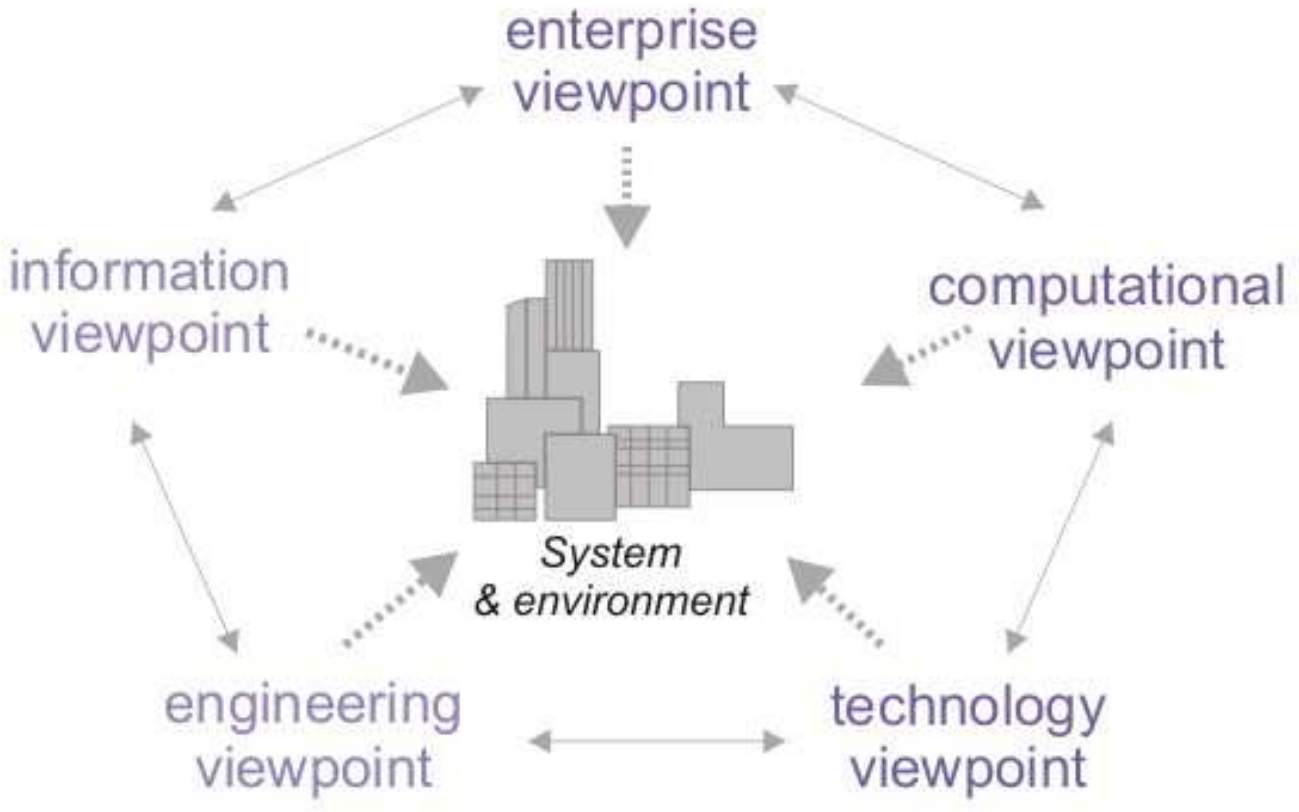


FIGURE 2.19 – Les différents points de vue définis dans la plateforme *RM-ODP*

les architectures logicielles la notion d'aspect, présentée dans la programmation orientée aspect (Voir section 2.4.2). Alors, cette application essaie d'établir une séparation des préoccupations verticales, associées à un point de vue particulier, des préoccupations transversales qui coupent horizontalement plusieurs points de vue de l'architecture logicielle, et sous-entendu les modèles associés à ces points de vue. Ainsi, la solution proposée pour cette problématique est de définir des aspects architecturaux représentés par des modèles partagés et qui représentent juste une seule préoccupation. Ces aspects architecturaux peuvent être soit *intra-view* appliqués au sein d'une seule vue, ou *cross-view* appliqués sur plusieurs vues.
Dans [van Heesch et al., 2012], une plateforme pour la documentation des décisions architecturales a été proposée. Cette plateforme est conforme au standard *IEEE 42010* et définie quatre points de vue dédiés juste à documenter les décisions architecturales et tend à couvrir entièrement les préoccupations associées à ces décisions. Ces points de vue sont : le point de vue "*Decision Detail*", "*Descision Relationship*", "*Decision Stakeholder Involvement*", et le point de vue "*Decision Chronological*".
Dans [Hilliard et al., 2012], une infrastructure dénotée *MEGAF* (*MEGa modeling Architecture Frameworks*) est proposée afin de faciliter la tâche des architectes logiciels de composer et de réutiliser les plateformes d'architecture logicielle et leurs entités de première classe, en tant qu'ensemble de points de vue, d'intervenants, de préoccupations, etc. Alors, cette infrastructure offre aux architectes logiciels les outils nécessaires pour définir des plateformes d'architecture logicielle conforment au standard *IEEE 42010*, soit en créant depuis zéro les entités de première classe, en les composant depuis des entités créées auparavant en utilisant cette infrastructure, ou en réutilisant des entités déjà créées.

2.6 Conclusion

Dans ce chapitre, nous avons présenté une synthèse bibliographique portant sur l'application de la notion de vue et point de vue dans les différents domaines de l'informatique, comme en spécification des besoins,

modélisation des systèmes, programmation, et surtout en architecture logicielle.
Ici, nous avons présenté les travaux les plus importants en architecture logicielle à base de vues ou points de vue, comme les approches *"4+1" View Model* et son successeur l'approche *Rational Architectural Description Specification* (*Rational ADS*), le standard *ISO/IEC/IEEE 42010*, l'approche *Views and Beyond* (*V&B*), l'approche de *Rozanski* et *Woods*, le modèle *Siemens*, la plateforme *Zachman*, et la plateforme *RM-ODP*.
Dans le chapitre suivant, nous devons mener une analyse comparative entre les approches étudiées en architecture logicielle afin de déterminer leurs limites.

3

Analyse comparative et limitations

Se basant sur l'étude bibliographique présentée dans le chapitre précédent, on peut constater que l'histoire des points de vue, présentée sous différents noms comme les vues, les perspectives, etc., est très riche ; en effet, beaucoup de contributions ont eu lieu dans différents domaines de l'informatique. Notre travail se situe dans le domaine des architectures logicielles multipoints de vue. Par la suite, nous effectuons une analyse comparative entre les différentes approches d'architectures logicielles.
La comparaison présentée dans le tableau 3.1, est basée sur des critères regroupés selon les quatre axes suivants :

- Trois critères sont été définis selon les objectifs et le public visé par l'approche :
  - L'objectif visé ;
  - Le noyau de l'approche ;
  - Les intervenants adressés.
- Plusieurs critères ont été définis selon le statut des vues de l'architecture logicielle et leur conformité à des styles architecturaux :
  - Statut/nombre de vues ;
  - Catégorisation des vues ;
  - Les styles architecturaux associés aux vues ;
  - La stratification des vues.
- Deux critères ont été définis selon l'aspect consistance entre les différentes vues de l'architecture logicielle :
  - L'intégration des vues ;
  - Les relations entre les vues.
- Finalement, un critère été défini selon la méthodologie proposée pour la définition et la construction de l'architecture logicielle :
  - Le processus de définition de l'architecture.

3.1 L'objectif visé

L'objectif visé représente les intentions, les buts, ainsi que les aspects couverts par l'approche. Ainsi :

- **le standard *IEEE 42010*** cherche la normalisation des concepts et pratiques liés à la description architecturale de vue d'un système à forte composante logicielle ;
- **le modèle *"4+1" View Model*** vise la description architecturale et la modélisation d'un système à forte composante logicielle guidée par les scénarios ;
- **l'approche *Rational ADS*** vise la description des systèmes informatiques complexes et des systèmes d'entreprise en définissant formellement l'évolution des besoins de ces systèmes et la testabilité de leurs architectures associées ;
- **l'approche *V&B*** vise la documentation d'une description architecturale qui soit déclinée en plusieurs vues s'adressant aux différents intervenants et répondant à leurs besoins afin d'assurer leurs communication ;
- **l'approche de *Rozanski et Woods*** vise la description de tous les aspects significatifs au niveau de l'architecture d'un système informatique en suivant une méthodologie bien précise éliminant les incohérences potentielles qui peuvent avoir lieu durant le processus de description ;
- **le modèle *Siemens*** vise la description des architectures logicielles suivant différents aspects significatifs au niveau de la modélisation et la construction du système intentionné ;
- **la plateforme *Zachman*** vise à organiser et classifier les différents modèles et artefacts d'une architecture d'un système informatique selon les différents aspects de développement et leur focus ;
- **le modèle référence *RM-ODP*** vise à standardiser le processus de développement des systèmes ayants des composants logiciels distribués et interagissant entre eux.

3.2 Le noyau de l'approche

Dans ce critère on vise le concept sur lequel l'approche est centrée. Ainsi :

- **le standard *IEEE 42010*** et **l'approche *V&B*** se basent sur les besoins des différents intervenants du système en construction ;
- les approches du **modèle *"4+1" View Model***, ***Rational ADS***, **le modèle *Simens***, et **lapproche de *Rozanski et Woods*** sont plutôt centrées sur les différentes phases de développement du système ;
- **la plateforme *Zachman*** est centrée sur les différentes phases de développement du système et l'organisation des modèles et des artefacts de chaque phase selon leurs focus.
- **le modèle *RM-ODP*** est centré sur la distribution des spécifications des systèmes informatiques.

3.3 Les intervenants adressés

Chaque approche d'architectures logicielles est adressée pour répondre aux besoins de certains intervenants. Ainsi :

- **le modèle *"4+1" View Model***, **l'approche *Rational ADS***, **la plateforme *Zachman***, **le modèle *Siemens***, et **l'approche de *Rozanski et Woods*** sont concernés par les concepteurs, les programmeurs, les testeurs, les ingénieurs de déploiement, et surtout et avant tout les architectes logiciels ;

- **le modèle *RM-ODP*** est concerné seulement par les programmeurs et les rédacteurs de normes ;
- **le standard *IEEE 42010*** et **l'approche *V&B*** n'ont pas spécifié explicitement les intervenants adressés, puisqu'ils sont beaucoup plus généraux. Mais les concepts proposés dans ces approches peuvent être appliqués sur n'importe quel intervenant, qu'il soit un membre de l'équipe de développement ou un utilisateur finale du système.

3.4 Statut/nombre de vues

Par statut, on entend si les vues sont fixes et prédéfinies par les approches ou si elles peuvent être créées (par instanciation d'un méta-concept) par l'architecte. En effet, dans la plupart des approches en spécification des besoins et dans l'approche *VUML*, les vues représentaient les perspectives des différents utilisateurs du système informatique, d'où le nombre de ces vues était variables et dépendant du domaine d'application de ces systèmes. Dans d'autres approches comme le standard *UML* les vues représentaient des perspectives liées aux différentes phases de développement, d'où leur nombre était toujours fixe. Concernant les approches en architecture logicielle :

- dans **le standard *IEEE 42010***, le nombre des vues est indéfini. Ainsi, l'architecte peut définir des vues en fonction des intervenants concernés ;
- dans **le modèle *"4+1" View Model***, les vues sont fixes et leur nombre est cinq : la vue Logique (*Logical*), Développement (*Development*), Processus (*Process*), Physique (*Physical*), et Scénarios (*Scenarios*). Tous les projets ne sont pas obligés de spécifier les cinq vues mais éventuellement un sous-ensemble d'entre elles. Ce sous-ensemble est choisi par l'architecte ;
- dans **l'approche *Rational ADS***, l'architecte peut choisir un sous-ensemble d'un ensemble de vues fixe qui contient neuf vues distinctes qui sont : la vue Besoins non-fonctionnels (*Non-Functional Requirements*), Cas d'étude (*Use Case*), Domaine (*Domain*), Expérience utilisateur (*User Experience*), Logique (*Logical*), Processus (*Process*), Implémentation (*Implementation*), Déploiement (*Deployment*), et la vue Teste (*Test*) ;
- dans **l'approche *V&B***, de nouvelles vues peuvent être créées par combinaison de vues existantes mais l'architecte ne peut pas introduire de nouveaux styles ou de nouvelles catégories de styles (*Viewtypes*) ;
- dans **l'approche de *Rozanski et Woods*** l'ensemble des vues est fixé à sept vues : Contexte (*Context*), Fonctionnel (*Functional*), Information (*Information*), Concurrence (*Concurrency*), Développement (*Development*), Déploiement (*Deployment*), et Opérationnelle (*Operational*). L'architecte peut utiliser un sous-ensemble de cet ensemble et peut aussi définir d'autres vues qui n'ont pas été prédéfinies selon ces besoins ;
- dans **le modèle *Siemens***, l'ensemble des vues est fixe et contenant quatre vues : la vue Conceptuelle (*Conceptual*), Module (*Module*), Code source (*Code*), et Exécution (*Execution*) ;
- dans **la plateforme *Zachman***, les vues sont définies sous le nom de perspectives. Les perspectives définies sont : Contexte (*Scope*), Modèle métier (*Business model*), Modèle des systèmes d'information (*Information systems model*), Modèle de technologie (*Technology model*), Besoins détaillés (*Detailed specification*), et système actuel (*Actual system*) ;
- finalement dans **l'approche *RM-ODP*** les vues sont définies sous le nom de points de vue et ils sont aussi fixes : Entreprise (*Entreprise*), Information (*Information*), Computationnel (*Computational*), Ingénierie (*Engineering*), et Technologie (*Technology*).

3.5 Catégorisation des vues

Plusieurs approches d'architectures logicielles ont défini des catégories pour leurs vues.

- dans **le standard *IEEE 42010*** et **l'approche de *Rozanski et Woods*** les vues architecturales sont catégorisées dans des points de vue. Un point de vue est une spécification des conventions de construction et d'utilisation d'une vue. Une vue doit être conforme à un point de vue ;

- dans **l'approche *V&B***, les vues sont organisées dans des *viewtypes*. Un *Viewtype* représente la structure du système en termes d'ensemble d'éléments et de relations entre eux selon des conventions et notations définies dans différents Styles ;

- les approches du **modèle *"4+1" View Model***, **l'approche *Rational ADS***, **le modèle *RM-ODP***, **le modèle *Siemens***, et **la plateforme *Zachman*** n'ont pas défini un principe d'organisation explicite pour leurs vues.

3.6 Les styles architecturaux associés aux vues

Les styles architecturaux définis comment les systèmes peuvent être construit, et comment leurs composants logiciels peuvent être organisés et manipulés, et comment ces derniers peuvent communiquer entre eux. Parmi les styles architecturaux les plus utilisés dans la construction des systèmes informatiques, on peut citer le style "Client-Serveur", "*Components & Connectors*", etc. En effet, l'architecte logiciel doit toujours penser aux styles architecturaux qu'il va utiliser durant les premières étapes de définition de l'architecture. Cependant la majorité des approches d'architectures logicielles n'ont pas considéré les styles dans leur conception.

- dans les approche du **modèle *"4+1" View Model***, et ***Rational ADS*** la possibilité d'appliquer des styles aux différentes vues de l'architecture a été noté, comme le style orienté objet pour la vue logique et le style *Pipes and Filters* pour la vue processus, etc. Cependant ils n'ont pas formalisé l'application de ces styles et ils ont donné au concepteur une flexibilité absolue à choisir les styles architecturaux pour les vues ;

- **l'approche *V&B*** est la seule parmi les autres qui a explicitement et formellement associé les styles architecturaux aux différents *viewtypes* comme le style *Pipes and Filters* qui a été associé au *viewtype C&C* ;

- **toutes les autres approches** n'ont pas considéré des styles architecturaux pour les vues.

3.7 La stratification des vues

Nous entendons par la stratification des vues l'association de plusieurs niveaux d'abstraction à ces vues. En effet, la notion de niveaux d'abstraction n'a pas été sérieusement adoptée et utilisée conjointement avec la notion des points de vue.
À notre connaissance, parmi les approches d'architectures logicielles, seule **l'approche *V&B*** introduit (implicitement) des niveaux d'abstraction ou d'information. Ces niveaux sont considérés durant la première étape du processus durant laquelle l'architecte doit spécifier le niveau d'information pour chaque case du tableau construit (*detailed information, some information, overview information*).

3.8 L'intégration des vues

Afin d'avoir une architecture logicielle multipoints de vue cohérente, chaque approche d'architectures logicielles doit disposer d'un mécanisme opératoire qui permet à l'architecte d'intégrer les différents points de vue définis dans l'architecture.

- dans **le modèle *"4+1" View Model***, les quatre vues principales (*Logical*, *Development*, *process* et *Physical*) sont intégrées à travers une cinquième vue qui est la vue "*scénarios*", d'où la notation *"4+1" View Model* ;

- dans **l'approche *V&B***, l'intégration des vues est considérée dans la deuxième étape du guide, où le concepteur combine les vues qui lui paraissent proches l'une de l'autre et néglige les autres vues qui se concentrent sur les détails inclus dans d'autres vues ;

- dans **l'approche *Rational ADS*** et **le modèle *Siemens***, cette intégrité a été informellement assurée à travers des relations fixes qui ont été introduites dans ces modèles pour exprimer qu'une vue doit respecter les définitions apportées dans une autre vue ;

- dans **l'approche de *Rozanski et Woods*** l'intégrité des vues dans l'architecture a été assurée à la manière des approches *Rational ADS* et *Siemens*, en ajoutant des recommandations sous la forme de questions que l'architecte se pose à la suite de la création de chaque vue pour détecter les incohérences potentielles ;

- dans **le standard *IEEE 42010***, **la plateforme *Zachman***, et **le modèle *RM-ODP***, l'intégration des vues est laissée à la charge de l'architecte logiciel sans aucune assistance.

3.9 Les relations entre les vues

En architecture logicielle, d'après l'étude que nous avons menée, nous n'avons pas trouvé des approches qui ont adressé la problématique de définition des relations formelles entre les différents points de vue et la mise en place des contraintes entre les différents modèles appartenant aux différentes vues de l'architecture logicielle. En effet, les approches cherchent juste à assurer un minimum d'intégrité entre les vues de cette architecture. Prenons comme exemple, le standard *IEEE 42010*, où une définition générique et un peu floue d'une notion de correspondance, entre n'importe quels deux éléments de l'architecture logicielle, a été introduite.

3.10 Le processus de définition de l'architecture (ADP)

En réalité, une minorité, parmi les approches d'architectures logicielles, a proposé des processus de définition des architectures logicielles.

- ***Rozanski et Woods*** ont proposé dans leur approche un processus de définition d'architecture logicielle détaillé et exprimé sous forme d'un diagramme d'activité. Ils ont assumé que ce processus se situe entre les deux phases de spécification des besoins et la conception ;

- dans **l'approche *V&B***, *Clements et al.* ont proposé un guide de trois étapes pour assister l'architecte logiciel à définir et documenter l'architecture logicielle ;

- **les autres approches** n'ont pas proposé des processus de définition d'architecture, ils ont laissé cette tâche à la charge de l'architecte logiciel.

TABLE 3.1 – Un tableau récapitulatif de l'analyse comparative entre les différentes approches d'architectures logicielles

	Objectif visé	Centrée sur	Intervenants	Nb de vues	Catégorie des vues	Styles architecturaux	Intégration des vues	Abstraction	Relations inter-vues	ADP
IEEE 42010	Standarisation	Besoins des intervenants	Tous	Indéfini	Point de vue	-	-	-	Notion de correspondance	-
V&B	Communication	Besoins des intervenants	Tous	Indéfini	Viewtype	Formel	2^{eme} étape du guide	Niveau d'information	-	Guide
Rozanski	Description d'une architecture	Phases de développement	Équipe de développement	7	Point de vue	-	Relations informelles /Conseils	-	-	Explicite
"4+1"	Modélisation	Phases de développement	Équipe de développement	5	-	Informel	Vue Scénarios	-	-	-
ADS	Évolution des besoins/Testabilité	Phases de développement	Équipe de développement	9	-	Informel	Relations informelles	-	-	-
Simens	Implémentation	Phases de développement	Équipe de développement	4	-	-	Relations informelles	-	-	-
Zachman	Classification des modèles	Phases de développement/Focus	Équipe de développement	6	-	-	-	-	-	-
RM-ODP	Développement de systèmes distribués	Distribution des spécifications	Programmeurs/ Rédacteurs de normes	5	-	-	-	-	-	-

3.11 Les limitations des approches étudiées

Malgré les avantages apportés par les approches étudiées au chapitre précédent dans le domaine de l'architecture logicielle multipoints de vue en termes de séparation des préoccupations des systèmes informatiques, de réduction de la complexité, ces dernières souffrent cependant encore de plusieurs limitations dont les principales :

1. les approches existantes ont échoué à résoudre les complexités au sein d'une vue. En effet, les préoccupations primitives de très haut niveau associées à une vue particulière d'une architecture logicielle ne peuvent jamais être traitées directement dans un seul modèle associé à un seul niveau d'abstraction, qui contient tous les détails nécessaires pour mener l'implémentation. Pourtant, on a besoin toujours de développer plusieurs modèles dans plusieurs niveaux d'abstraction pour pouvoir répondre effectivement à ces préoccupations primitives. Ainsi, une hiérarchie plate d'une vue architecturale dans laquelle on peut trouver les modèles de très haut niveau à côté des modèles très proches de l'implémentation est toujours inconfortable pour les architectes logiciels. Alors, nous pensons que pour réduire ces complexités on pourrait organiser les modèles associés aux vues dans une hiérarchie de plusieurs niveaux d'abstraction ;

2. peu d'approches proposent un processus de définition d'une architecture logicielle. Pourtant, la tâche de définition d'architectures logicielles est une tâche vraiment lourde, critique et difficile. Pour cela, un architecte logiciel a toujours besoin de meilleures pratiques rassemblées dans une méthodologie ou un processus de définition, afin qu'il soit capable de construire effectivement, et dans une durée acceptable, une architecture logicielle appropriée répondant aux problèmes visés par le système informatique. Par ailleurs, les processus proposés n'intègrent pas parmi leurs activités, la définition de niveaux de description ;

3. les approches existantes n'ont pas résolu les problèmes d'inconsistance qui peuvent avoir lieu entre les différentes vues de l'architecture, même si la plupart de ces approches ont admis l'existence de certains aspects complémentaires et dépendants entre les différentes vues de l'architecture logicielle. Ces problèmes d'inconsistance résultent du fait que les préoccupations du système informatique seront traitées séparément dans plusieurs vues de l'architecture logicielle, sans que les intervenants impliqués dans la construction d'une vue donnée de cette architecture, seront impliqués dans la construction des autres vues.

3.12 Conclusion

Le but de ce chapitre est de déterminer les limitations des approches existantes en architecture logicielle qui portent sur l'introduction de la notion de vue au sein de ces architectures. Ainsi, pour pouvoir déterminer ces limitations, nous avons mené une analyse comparative entre ces approches en considérant neuf critères différents : (1) l'objectif visé, (2) le noyau de l'approche, (3) les intervenants adressés, (4) le statut/nombre de vue, (5) la catégorisation des vues, (6) les styles architecturaux associés aux vues, (7) la stratification des vues, (8) l'intégration des vues, (9) les relations entre les vues, et (9) le processus de définition de l'architecture.
Les limites identifiés dans ce chapitre vont être la base des motivations et des objectifs d'une nouvelle approche d'architecture logicielle multipoints de vue.

MoVAL
(Model, View, and Abstraction Level based software architecture)

4

MoVAL : Approche et concepts de base

4.1 Introduction

Dans les chapitres précédents, nous avons effectué une synthèse bibliographique portant sur les utilisations de la notion de point de vue dans la littérature informatique, sous ses différentes dénotations comme les perspectives ou les vues. Ainsi, nous avons présenté les approches basées sur cette notion dans les domaines de spécification des besoins, de modélisation des systèmes, de programmation et en architecture logicielle. Ensuite nous avons mené une analyse comparative sur les approches à base de point de vue en architecture logicielle afin de déterminer leurs limitations.
L'objectif de ce chapitre est de présenter une nouvelle approche d'architecture logicielle hiérarchique multipoints de vue qui répond aux limitations des approches existantes présentées au chapitre précédent.
Au début, nous commençons par la définition des objectifs de notre travail et notre positionnement par rapport aux autres travaux. Ensuite, nous présentons un cas d'étude d'un système Web de ventes électronique (SWVE) sur lequel nous allons appliquer notre approche pour démontrer sa validité. Après, nous présentons notre approche *MoVAL* (*Model, View, and Abstraction level based software architecture*), et nous définissons ses concepts de base et les extensions apportées. Puis, nous présentons successivement le méta-modèle, la configuration architecturale et un catalogue de points de vue que nous avons proposé.

4.2 Positionnement et motivations de notre approche

Notre travail se situe dans le cadre des études sur les architectures logicielles et la proposition d'une approche d'architecture logicielle hiérarchique multipoints de vue, qui a comme but de répondre aux limitations soulignées au paragraphe 3.11 du chapitre précédent et que nous représentons ici :

1 la complexité au sein des vues ;

2 l'absence d'un processus de définition d'architecture complet ;

3 les inconsistances inter-vues.

Pour atteindre les objectifs ci-dessus, nous adoptons une stratégie basée sur le concept de la décomposition, qui est une stratégie utilisée essentiellement pour organiser un système dans un nombre de parties afin de réduire sa complexité en termes de conception, modularité, maintenance, etc. ... . Pour aboutir à une application efficace du concept de décomposition, il est indispensable de minimiser toujours les dépendances

entre les différentes parties obtenues et améliorer leurs cohésions.
En effet, cette stratégie est utilisée déjà dans plusieurs domaines de l'informatique, comme l'analyse structurée décrite par *Tom DeMarco* [DeMarco, 1979], la modularité en programmation orientée objet et en modélisation orientée objet, etc. ...
Ainsi, dans *MoVAL* la décomposition est adoptée selon les trois dimensions suivantes :

- la décomposition de la description d'architecture en plusieurs vues ;
- la décomposition des détails de modélisation dans une vue donnée suivant plusieurs niveaux d'abstraction ;
- la décomposition du processus de définition d'architecture selon plusieurs phases.

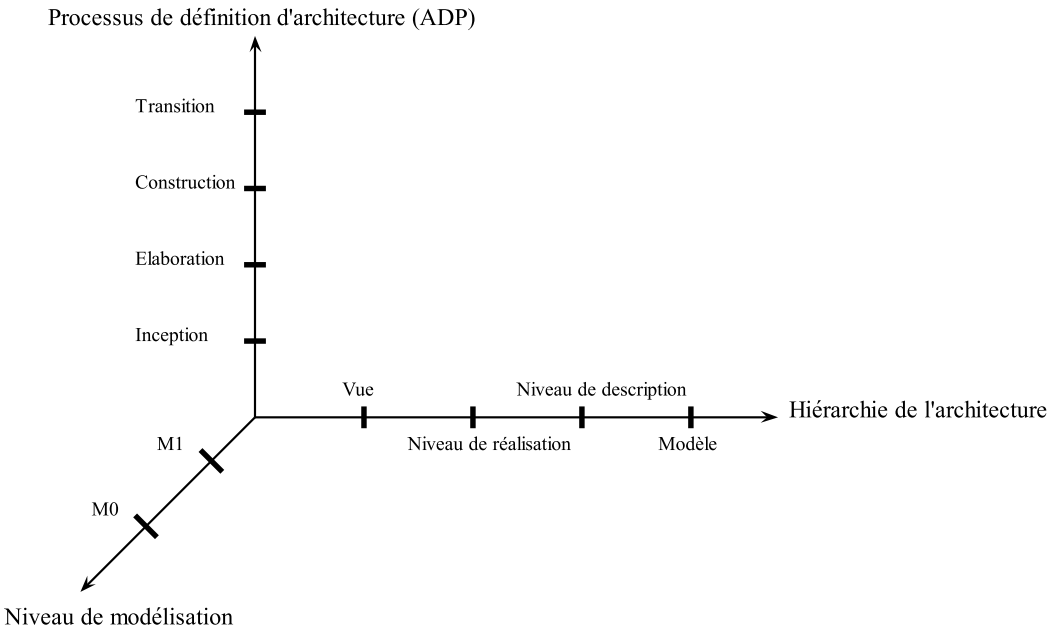


FIGURE 4.1 – Triptyque représentant les intentions de notre approche

Le triptyque de la figure 4.1 illustre les intentions de notre approche selon trois axes qui sont :

- l'axe "*Hiérarchie de l'architecture*" illustre les types d'entités qui composent l'architecture, à savoir une *vue*, un *niveau de réalisation*, un *niveau de description* et un modèle.
- l'axe "*Processus de définition d'architecture*", noté *ADP*, présente les quatre différentes phases composantes du processus proposé dans notre approche, qui sont : (1) la phase "*Inception*", la phase "*Elaboration*", la phase "*Construction*" et la phase "*Transition*".
- l'axe *Niveau de modélisation* illustre le fait que notre approche adresse les besoins, les exigences, et les préoccupations des différents intervenants dans les différents niveaux de modélisation : l'architecte logiciel, les analystes, et les concepteurs au niveau de modélisation *M1*, les architectes des applications (les ingénieurs de déploiement) et les utilisateurs au niveau de modélisation *M0*.

Le tableau 4.1 présente l'analyse comparative du chapitre 3 incluant une nouvelle ligne pour l'approche *MoVAL* que nous proposons.

TABLE 4.1 – Un tableau récapitulatif de l'analyse comparative entre les différentes approches d'architectures logicielles

	Objectif visé	Centrée sur	Intervenants	Nb de vues	Catégorie des vues	Styles architecturaux	Intégration des vues	Abstraction	Relations inter-vues	ADP
IEEE 42010	Standarisation	Besoins des intervenants	Tous	Indéfini	Point de vue	-	-	-	Notion de correspondance	-
V&B	Communication	Besoins des intervenants	Tous	Indéfini	Viewtype	Formel	2^{eme} étape du guide	Niveau d'information	-	Guide
Rozanski	Description d'une architecture	Phases de développement	Équipe de développement	7	Point de vue	-	Relations informelles /Conseils	-	-	Explicite
"4+1"	Modélisation	Phases de développement	Équipe de développement	5	-	Informel	Vue Scénarios	-	-	-
ADS	Évolution des besoins/Testabilité	Phases de développement	Équipe de développement	9	-	Informel	Relations informelles	-	-	-
Simens	Implémentation	Phases de développement	Équipe de développement	4	-	-	Relations informelles	-	-	-
Zachman	Classification des modèles	Phases de développement/Focus	Équipe de développement	6	-	-	-	-	-	-
RM-ODP	Développement de systèmes distribués	Distribution des spécifications	Programmeurs/ Rédacteurs de normes	5	-	-	-	-	-	-
MoVAL	**Description d'une architecture**	**Besoins des intervenants et une description à multi-granularité**	**Tous**	Indéfini	**Point de vue**	**Informel**	**Liens inter-vues**	**Niveau de rálisation/Niveau de description**	**Liens architecturaux**	**Explicite**

4.3 SWVE : notre cas d'étude

Afin de valider notre approche, un exemple d'un système web de ventes électroniques, abrégé *SWVE*, va être considéré afin de clarifier et concrétiser les éléments de base et la contribution apportée par l'approche *MoVAL*. En effet, cet exemple a été introduit dans le livre de *Rozanski et Woods* [Rozanski and Woods, 2011] pour illustrer la notion d'une vue fonctionnelle. Nous l'utiliserons ici suite à des changements et des extensions majeurs afin de lui permettre d'encapsuler toutes les notions voulues.
Le *SWVE* est un système qui consiste en trois applications principales, la première est dédiée pour les clients afin de pouvoir effectuer les ventes électroniques, la deuxième est une application d'administration qui permet aux administrateurs de manipuler la liste des produits introduits dans le système, enfin la troisième application est dédiée pour les services clientèles. Tous les éléments de ce système appartiennent à un seul réseau *LAN*, sauf les exécutions des commandes seront effectuées sur un composant extérieur.
La structure physique de ce système est représentée dans la figure 4.2. Dans cette figure, les composants du système *SWVE*, comme les applications web, les applications bureautiques, les serveurs et les bases de données sont représentés par des images spécifiques à chaque type de composant. Les relations entre ces éléments sont représentées par des lignes droites les rejoignant.

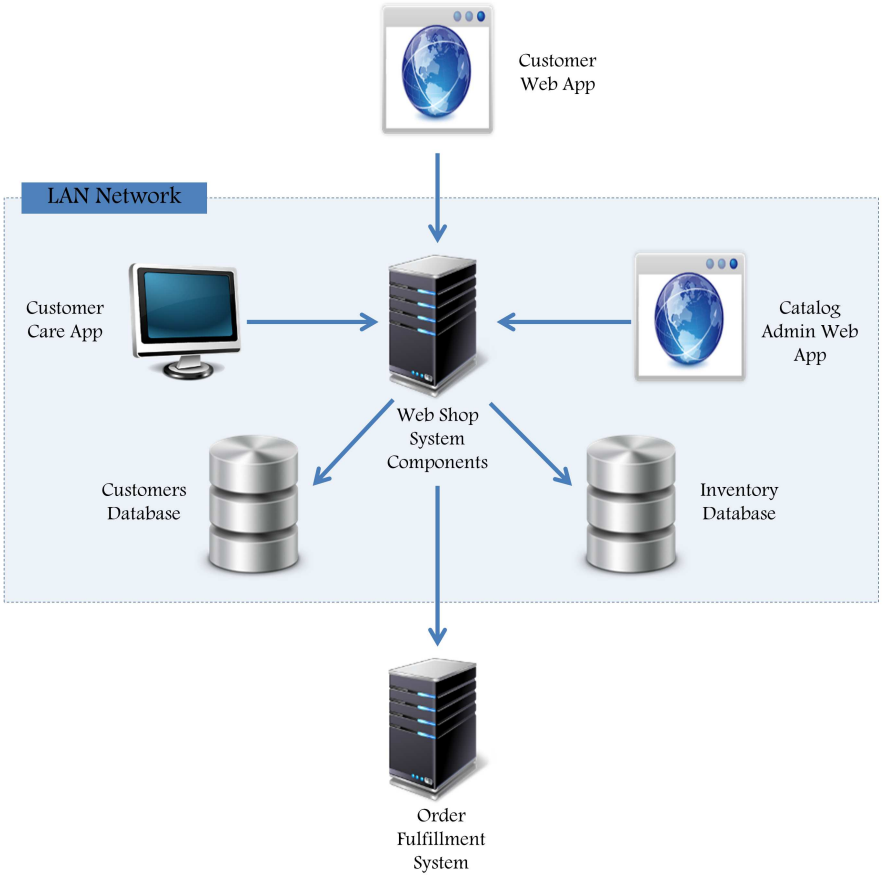


FIGURE 4.2 – Structure physique de SWVE.

La figure 4.3 représente les différents composants logiciels offrant les fonctionnalités du système *SWVE* à travers une notation informelle, dite "*Boxes-and-lines*". Ensuite, les figures 4.4 et 4.5 représentent ces composants d'une manière un peu plus formelle et technique à travers des diagrammes *UML* de composants à deux niveaux de détails.

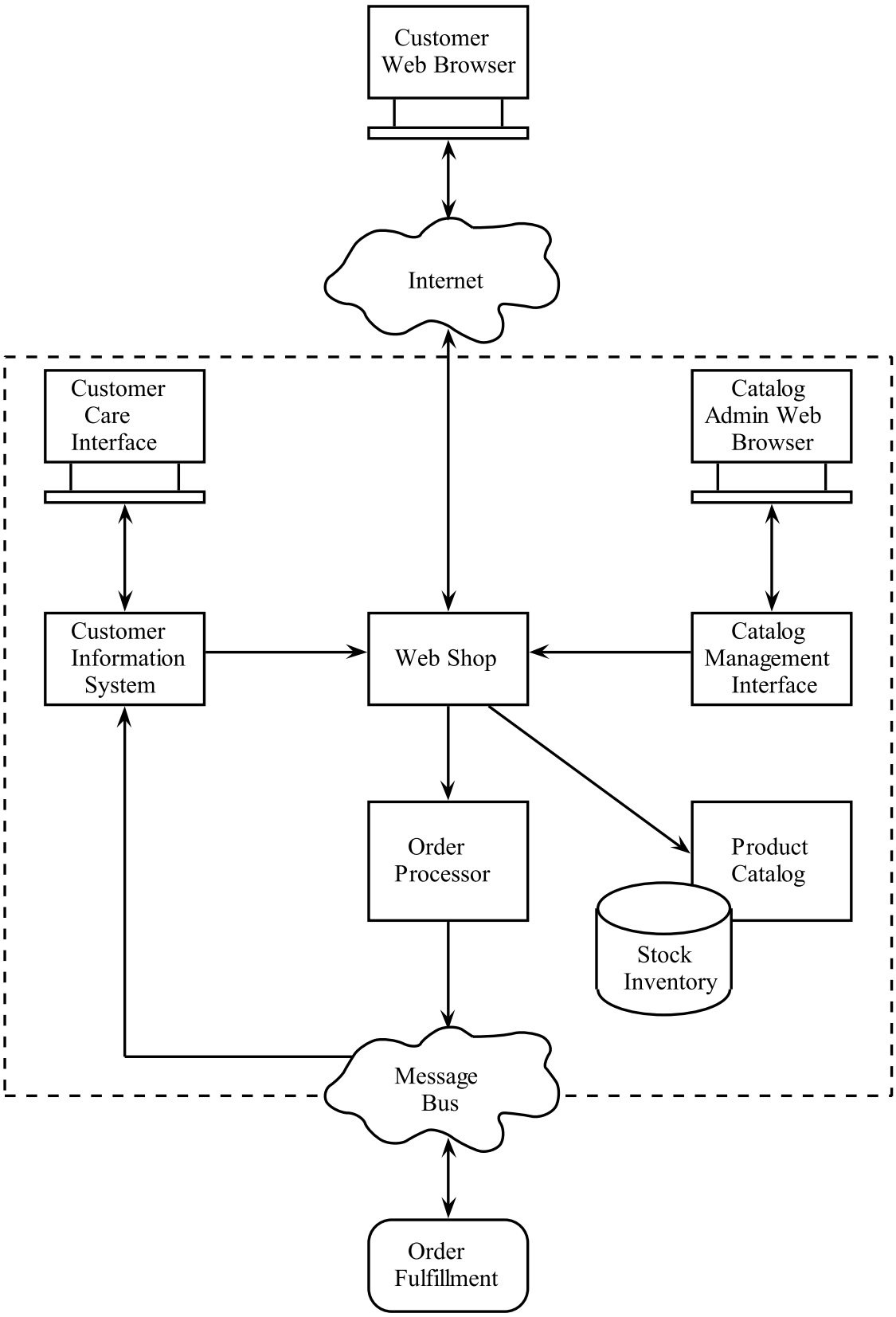


FIGURE 4.3 – Un modèle informel représentant les fonctionnalités du *SWVE*.

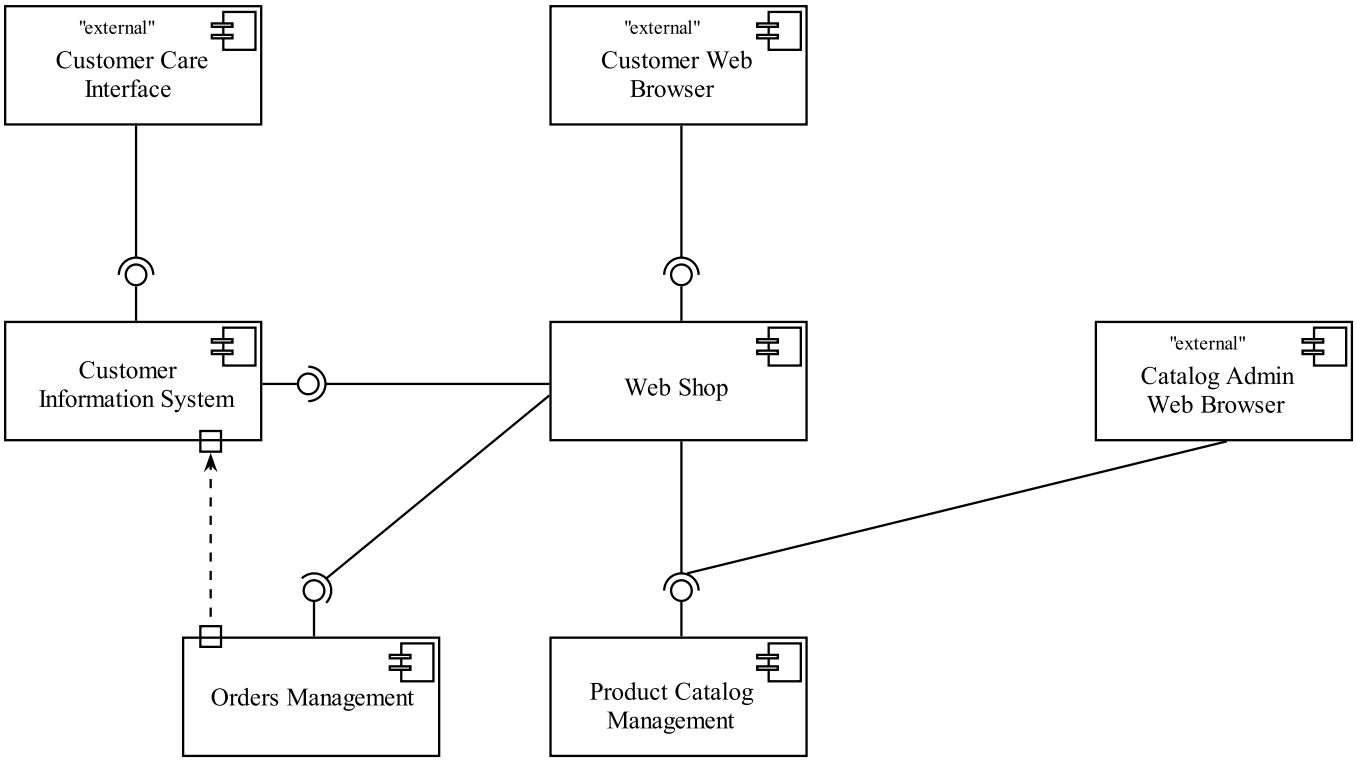


FIGURE 4.4 – Un modèle de composants représentant les fonctionnalités du *SWVE*.

4.4 MoVAL et les standards

L'approche *MoVAL* (*Model, View, and Abstraction Level based software architecture*), comme le nom indique, est une approche d'architecture logicielle hiérarchique multi-vues qui vise l'organisation des modèles des systèmes informatiques dans des hiérarchies à base de vues et de niveaux d'abstraction, afin de réduire la complexité de conception et de développement de ces systèmes, et afin de permettre une meilleure communication entre les intervenants concernés.
En effet, cette approche se vient être conforme aux standards les plus adoptés et les plus utilisés dans le domaine des architectures logicielles et de développement informatique. Ainsi, elle est conforme à trois standards très réputés qui sont :

- le standard *ISO/IEC/IEEE 42010* [ISO/IEC/IEEE, 2011]
  Ce standard a été conçu par l'*APG* (*The IEEE Architecture Planning Group*) afin de standardiser la définition d'une architecture logicielle multipoints de vue et ses éléments principaux.
  MoVAL hérite les définitions des éléments principaux de ce standard comme les définitions d'une architecture logicielle, d'un point de vue, d'une vue, d'un modèle, etc. et les étend par de nouvelles notions qui seront présentées plus loin ;

- le standard *MOF* (*Meta-Object Facility*) [OMG, 2013]
  Ce standard est conçu par l'*OMG* (*Object Management Group*). Il s'intéresse à la représentation des méta-modèles et leur manipulation. Le standard *MOF* se situe au sommet d'une architecture de quatre couches : le méta-méta-modèle *MOF* (*M3*) ; le méta-modèle (*M2*) ; le modèle (*M1*) ; et l'application (*M0*).
  En effet, le méta-modèle de *MoVAL* se situe au niveau *M2*, et permet aux équipes de développement de créer des modèles ou des architectures pour des systèmes informatiques au niveau *M1*, qui peuvent être implémentées par la suite au niveau *M0* ;

- le standard *MDA* (*Model Driven Architecture*) [Miller and Mukerji, 2003]
  C'est une architecture dirigée par les modèles, offrant une stratégie ou une démarche de réalisation

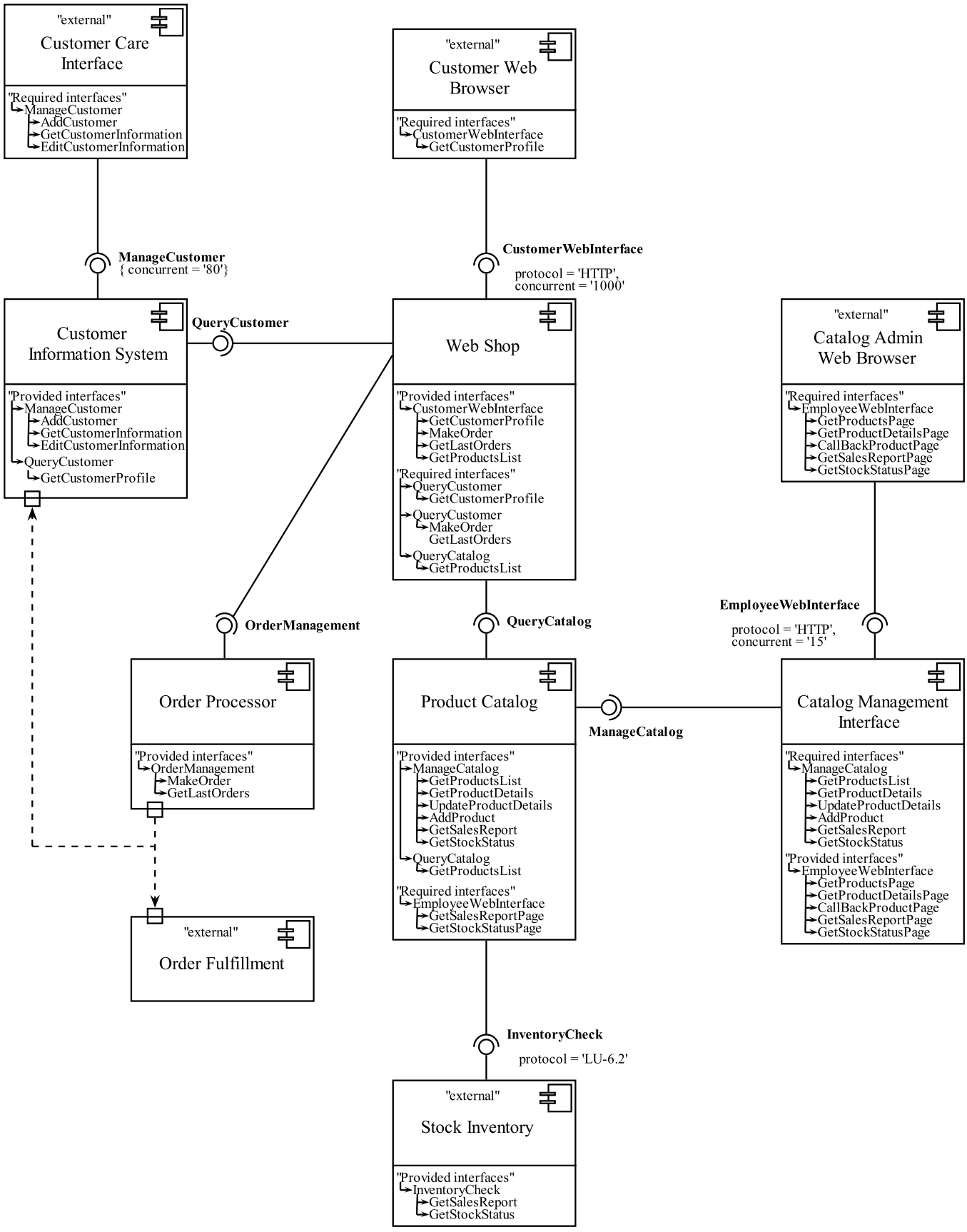


FIGURE 4.5 – Un modèle de composants detaillé représentant les fonctionnalités du *SWVE*.

des systèmes informatiques. Cette architecture a été proposée et soutenue par l'*OMG* (*Object Management Group*). Le principe de cette architecture est d'élaborer les différents modèles d'un système informatique en partant d'un ensemble de modèles métiers indépendants de l'informatisation, qui seront transformés en premier lieu en des modèles indépendants de la plateforme, puis en des modèles spécifiques à la plateforme cible pour l'implémentation concrète du système.
En effet, *MoVAL* adopte cette stratégie en permettant et en recommandant aux architectes logiciels de définir plusieurs niveaux de réalisation qui peuvent soutenir les trois niveaux définis au sein du standard *MDA*.

4.5 Définitions de notions de base

Comme l'approche *MoVAL* est conforme à plusieurs standards, elle hérite de ces standards les définitions de plusieurs concepts de base, comme la définition d'une architecture logicielle, une description d'architecture, un point de vue, une vue et un modèle.

4.5.1 Architecture logicielle, description d'architecture et style architectural

Un système informatique peut être assimilé, selon le standard IEEE 42010 [ISO/IEC/IEEE, 2011], à une combinaison d'éléments logiciels en relation les uns avec les autres formant un tout. Il peut être assimilé aussi à un ensemble de sous-systèmes formant une unité, qui agissent et interagissent conjointement pour l'accomplissement d'une finalité.
L'organisation de ces éléments logiciels ou de ces sous-systèmes est dénotée une architecture logicielle. Cette architecture logicielle regroupe les différents éléments ou sous-systèmes d'un système informatique, et représente leurs interrelations et leurs interactions. L'architecture logicielle représente également comment ce système doit être conçu pour répondre aux spécifications.
Habituellement, une architecture logicielle est construite conformément à un style architectural spécifique qui répond aux exigences architecturales du système selon son domaine d'application et l'environnement avec lequel il interagit. Ce style architectural, parfois appelé un parton architectural, représente un ensemble de principes qui forment une plateforme abstraite de construction d'une famille de systèmes informatiques. Normalement, l'importance et l'avantage ultime des styles architecturaux est la réutilisation des solutions conceptuels des problèmes architecturaux les plus récurrents. Parmi les styles architecturaux les plus connus on peut mentionner le style *Client/Serveur*, *Layered Architecture*, *3-Tier/N-Tier*, etc. ...
En effet, une description d'architecture (*i.e. Architectural Description*), est un artefact qui décrit l'architecture d'un système donné. Cette description d'architecture doit identifier et décrire les informations cruciales d'une architecture, comme les intervenants et leurs préoccupations ou exigences et leurs représentations dans des modèles afin de résoudre les problèmes structurellement significatifs.
A titre d'exemple, l'architecture logicielle du système *SWVE*, introduit auparavant, est l'organisation de ses trois applications sur un réseau *LAN* et leurs interactions avec des composants locaux et extérieurs. Cependant la description d'architecture est la description de cette organisation par des diagrammes spécifiques.

4.5.2 Vue et point de vue

Partant du fait qu'un système peut être analysé selon différentes vues (sous différents angles), nous pourrons dire qu'une architecture d'un système est constituée, en général, d'un ensemble de vues distinctes reflétant ses différents aspects. Les vues les plus couramment utilisées, lors de la phase de modélisation, sont la vue structurelle et la vue comportementale. Cependant, ceci n'interdit pas d'autres choix, tels que la prise en compte de l'aspect physique ou même thermique dans le cadre d'un système d'automobile par exemple. Ainsi, la liste des vues à établir n'est pas figée, bien au contraire, elle évolue et dépend directement du type de problèmes étudiés ainsi que du domaine d'application.

Les architectures logicielles vues sous cet angle, peuvent être appelées des architectures logicielles multi-points de vue.
Le standard IEEE 42010, auquel l'approche *MoVAL* est conforme, distingue entre les notions de vues et de points de vue. En effet, ce standard définit un point de vue comme étant, d'une part, un regroupement des conventions spécifiant les formalismes de modélisation, les méthodes de modélisation, les techniques d'analyse, et toute autre opération nécessaire pour construire une vue d'une architecture logicielle.
Dans MoVAL, un point de vue est défini par la liste des intervenants qui lui sont associés, les préoccupations ou exigences couvertes, et par ses formalismes. Un formalisme est un langage de modélisation ou de construction d'un artefact, ou un ensemble d'entités et de notations graphiques ayant des sémantiques bien définies et non ambigues, permettant la construction de diagrammes suffisamment clairs et sans ambiguité. On peut citer comme exemples de formalismes les diagrammes de cas d'utilisation, diagrammes de composant, diagrammes de classe, etc. qui ont été définis dans *UML* (*Unified Modeling Language*).
Dans MoVAL, une vue est une entité qui compose les descriptions d'architectures logicielles, et qui est conforme à un ou plusieurs points de vue dans le sens qu'elle utilise les conventions et formalismes dans les opérations de construction des modèles de cette vue.
Revenons au système *SWVE*, on peut lui définir une vue fonctionnelle et une vue physique. La vue fonctionnelle est associée à un point de vue fonctionnel englobant les préoccupations concernant les fonctionnalités offertes par le système, et acceptant le formalisme diagramme *UML* de composant. Les figures 4.3,4.4, et 4.5 représentent des modèles appartenant à cette vue. La vue physique est associée à un point de vue physique englobant les préoccupations des ingénieurs de réseaux et de déploiement. Ensuite, un formalisme peut être associé à ce point de vue. Les éléments de ce formalisme sont des images illustrant les différents types de composants, comme les applications web, les applications bureautiques, les serveurs, les bases de données, etc., les relations entre ces éléments sont des lignes droites rejoignant les deux éléments. La figure 4.2 représente un exemple de modèle appartenant à la vue physique, qui est construit à travers le formalisme décrit au-dessus.

4.5.3 Les modèles

En général, les modèles peuvent être considérés comme étant les composants ou les éléments les plus importants dans une description d'architecture logicielle et cela revient au fait qu'ils représentent le moyen de communication le plus simple et efficace entre l'architecte logiciel d'une part, et les intervenants ayant des intérêts dans le système d'une autre part, pour que ces derniers puissent comprendre et analyser les situations que le premier essaie de modéliser. Dans d'autres cas, ces modèles représentent le moyen de discussion le plus formel et le plus précis entre les différents intervenants du système ayant quelque part des intérêts imbriqués. En outre, les modèles organisent les processus de développement logiciel à travers l'organisation des différentes tâches et leur affectation aux équipes de développement en spécifiant ces modèles en tant que artefacts de sortie attendus pour chaque tâche.
Alors, un modèle devra être communiqué par l'architecte logiciel aux intervenants appropriés afin d'illustrer formellement et sans ambiguité la structure de la solution apportée par le premier en ce qui concerne les exigences associées, et qui doivent être couvertes et respectées par ces intervenants. Ainsi, par le biais des modèles l'architecte logiciel peut diffuser sa vision vis-à-vis de l'architecture et de la structure du système en considération, et il peut de cette manière déclarer les détails qui lui paraissent structurellement significatifs pour guider le développement de ce système suivant le plan prévu.
Normalement, un modèle doit être construit suivant un formalisme bien défini et bien acceptable dans le contexte du point de vue et du niveau d'abstraction où il se trouve, afin de présenter et de détailler les informations pertinentes pour la construction du système en considération dans le même contexte. Après, ce modèle peut être lié à d'autres modèles appartenant à son niveau d'abstraction par des liens de consistance, afin d'assurer la cohérence des sémantiques représentées au sein de l'ensemble des modèles de ce niveau. En fait, ces liens, marqués "is_R", peuvent soutenir plusieurs types de relation comme la composition et l'agrégation.

En effet, *MoVAL* est une approche d'architecture logicielle dirigée par les modèles, qui est conforme avec le *MDA* (*Model Driven Architecture*) [Miller and Mukerji, 2003]. Le *MDA* est une démarche de développement logiciel, proposée et soutenue par l'*OMG*. Cette démarche se base sur l'élaboration de différents types de modèles, en partant des modèles métiers indépendants de l'informatisation (*Computation Independent Model*, *CIM*), puis la transformation de ces modèles en d'autres modèles indépendants de la technologie utilisée (*Platform Independent Model*, *PIM*), et enfin la construction à partir des *PIM* d'autres modèles spécifiques à une technologie ou une plateforme particulière (*Platform Specific Model*, *PSM*) afin de pouvoir concrètement implémenter le système.

- Un modèle indépendant de l'informatisation (*Computation Independent Model*, *CIM*), est un modèle dans lequel l'architecte logiciel se focalise sur le contexte du système en considération, et sur les exigences qui sont en relation surtout avec l'environnement dans lequel le système agit. Ce type de modèle a pour but de raccourcir les distances entre les différents intervenants et l'architecte logiciel responsable de la construction d'une architecture qui répond aux besoins des premiers.

- Un modèle indépendant de la plateforme (*Platform Independent Model*, *PIM*), est un modèle dans lequel on se focalise sur la représentation du système sous l'optique d'un ensemble d'exigences ou de préoccupations. Dans ce type de modèles, on ne s'intéresse jamais des détails d'implémentation en accord avec une technologie de développement spécifique.

- Un modèle spécifique à la plateforme (*Platform Specific Model*, *PSM*), est un modèle qui prend en considération tous les détails d'implémentation et surtout celles en accord avec la plateforme cible. Alors il représente un ensemble d'exigences du système dans le cadre d'une plateforme de développement particulière. Ainsi, ces modèles doivent être proches d'une implémentation concrète du système.

En effet, l'approche *MoVAL*, a été conçue conforme à ce standard *MDA* afin de laisser la porte ouverte au développement d'outils de génération automatique de code source à partir de la définition de l'architecture du système.

4.6 Les extensions apportées par *MoVAL*

En effet, l'approche *MoVAL* étend les concepts de base hérités des standards adoptés par de nouvelles notions, comme la notion de niveaux d'abstraction et la notion de liens architecturaux.

4.6.1 Niveau d'abstraction d'une architecture

Malgré les possibilités qu'offre l'approche multi-vue ou l'architecture multipoints de vue, il est courant que l'élaboration d'une vue donnée d'une architecture logicielle soit trop complexe pour être mise en oeuvre facilement. Par ailleurs, afin de maîtriser cette complexité d'une façon graduelle, il est indispensable d'adopter une approche hiérarchisée qui fasse apparaître différents niveaux de compréhension, ou niveaux d'abstraction, dans une vue.
L'abstraction elle-même est une notion abstraite ; c'est la relégation de certains détails qui paraissent inutiles, et le fait de se focaliser sur les aspects les plus importants, à un instant donné, du système. Cette définition a été adoptée par de nombreux travaux, comme ceux de *Kramer* [Hazzan and Kramer, 2007], *Regnell et al.* [Regnell et al., 1996], *Medvidovic et al.* [Medvidovic et al., 1996], et *Momperrus et al.* [Monperrus et al., 2009].
Dans MoVAL, deux types d'abstraction, ou en d'autres mots deux dimensions d'abstraction, ont été définies, qui sont :

(1) l'abstraction de réalisation, dénotée niveau de réalisation,

(2) et l'abstraction de description, dénotée niveau de description.

Le niveau de réalisation

De nature incrémentale, l'activité de définition d'architecture logicielle passe à travers plusieurs phases. A la fin de chaque phase, l'architecte doit élever le niveau de concrétisation de son architecture. En conséquence, ce dernier doit créer des modèles et des artefacts qui répondent mieux à une question essentielle, qui est "*Comment ?*" ou "*How to ?*". Par exemple, si l'architecte logiciel a fourni, pendant une phase donnée du processus de définition de l'architecture logicielle, des modèles de cas d'utilisation illustrant les fonctionnalités couvertes par le système, il doit se poser, durant les phases les plus avancées du processus de définition de l'architecture, la question "Comment implémenter ces cas d'utilisation ?", et la réponse à cette question sera en fournissant des modèles plus proches de l'implémentation comme les modèles à composants. Puis, dans d'autres phases encore plus avancées, il se pose une autre question "Comment mettre ces composants en production ?", et la réponse peut être fournie à travers d'autres modèles comme les modèles de déploiement.
La figure 4.6 illustre la notion de concrétisation en représentant trois modèles différents *n*, *n-1*, et *n-2* qui sont construits à travers trois formalismes distincts. Le modèle du niveau *n* est le modèle le moins concret, et alors ayant le niveau de réalisation inférieur, et le modèle du niveau *n-2* est le modèle le plus concret et alors ayant le niveau de réalisation supérieur.

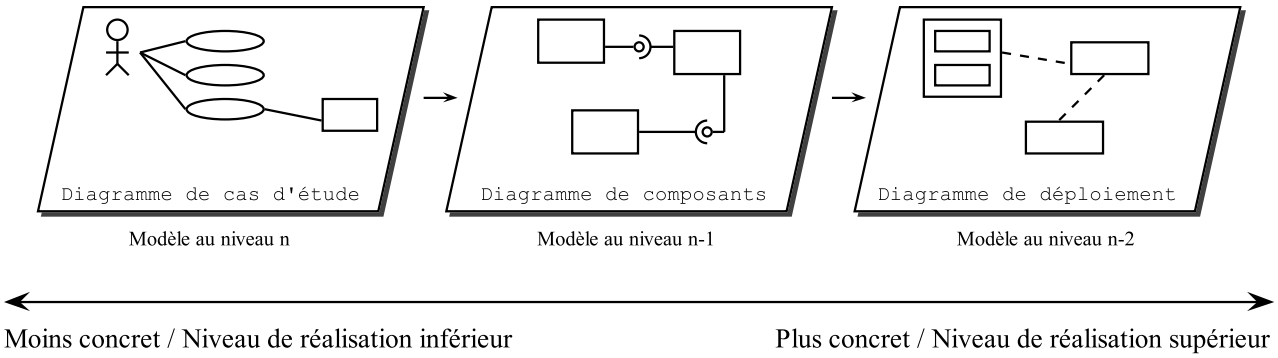


FIGURE 4.6 – La notion de concrétisation dans les niveaux de réalisation.

Selon *MoVAL*, un niveau de réalisation est conforme à un point de vue particulier. De plus, le passage d'un niveau de réalisation à un autre niveau plus concret, implique l'existence d'un lien de type is_{+A}. La figure 4.7 représente deux niveaux de réalisation n1 et n2 d'une architecture logicielle *MoVAL*. Ainsi, n2 étant plus concret que n1, alors n2 est lié par un lien vertical is_{+A} à n1.

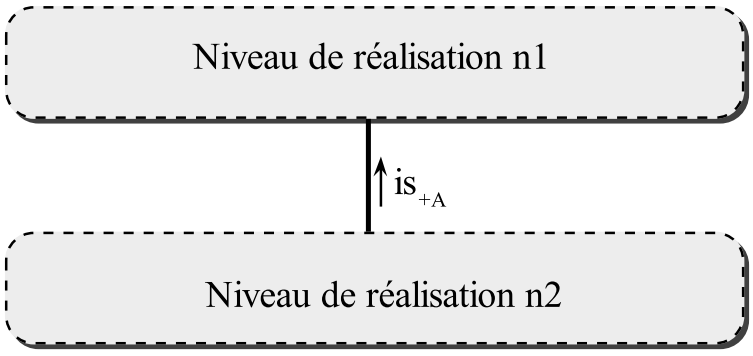


FIGURE 4.7 – Le lien is_{+A} entre deux niveaux de réalisation.

En général, le passage d'un niveau de réalisation à un autre implique toujours un passage d'un point de vue à un autre, à condition que ce point de vue soit toujours associé à la vue impliquée. Cela revient au fait que ce passage a eu lieu suite à un changement des préoccupations ou/et des formalismes considérés dans le niveau précédent. Ainsi, un niveau de réalisation doit respecter et obéir aux formalismes impliqués dans

le nouveau point de vue.
Il est à noter que la transition du niveau de réalisation n1 au niveau n2 indique une transition d'une phase dans le processus de développement, ou le processus de définition de l'architecture logicielle, à une autre phase plus avancée. Par exemple, dans le système *SWVE*, l'architecte logiciel peut définir, dans un niveau de réalisation, les fonctionnalités offertes par le système à travers un diagramme informel comme celui de la figure 4.3, puis il définit dans un niveau de réalisation supérieur comment implémenter ces fonctionnalités à travers un modèle de composant, comme celui de la figure 4.4.

Le niveau de description

Le deuxième type d'abstraction défini dans l'approche *MoVAL* est l'abstraction de description. Ce type d'abstraction permet à l'architecte logiciel d'effectuer des *Zoom In/Zoom Out* sur les modèles associés à une vue de l'architecture logicielle. En effet, ce type d'abstraction permet de fournir une multitude de niveaux de description allant d'une granularité "*Coarse-grained*" à une granularité "*Fine-grained*".
Dans le cas de la granularité "*Coarse-grained*", l'architecte logiciel définit des modèles de haut-niveau représentant la structure globale sans avoir besoin de fournir des détails supplémentaires. Comme par exemple, dans une structure à composants, la représentation juste des composants de haut-niveau sans fournir les détails décrivant la composition de chacun de ces composants composites.
Dans le cas de la granularité "*Fine-grained*", l'architecte logiciel fournit tous les détails, comme dans l'exemple précédent, il fournit la composition de bas-niveau de chaque composant composite. La figure 4.8 illustre un exemple sur les niveaux de granularité, ou les niveaux de description, de trois modèles différents.
La figure 4.8 illustre la notion de granularité en représentant trois modèles différents *n*, *n-1*, et *n-2* qui ont des niveaux de granularité différents. Le modèle du niveau n est le modèle ayant le niveau de granularité inférieur, ainsi ayant le niveau de description inférieur, et le modèle du niveau *n-2* est le modèle ayant le niveau de granularité supérieur et alors ayant le niveau de description supérieur.

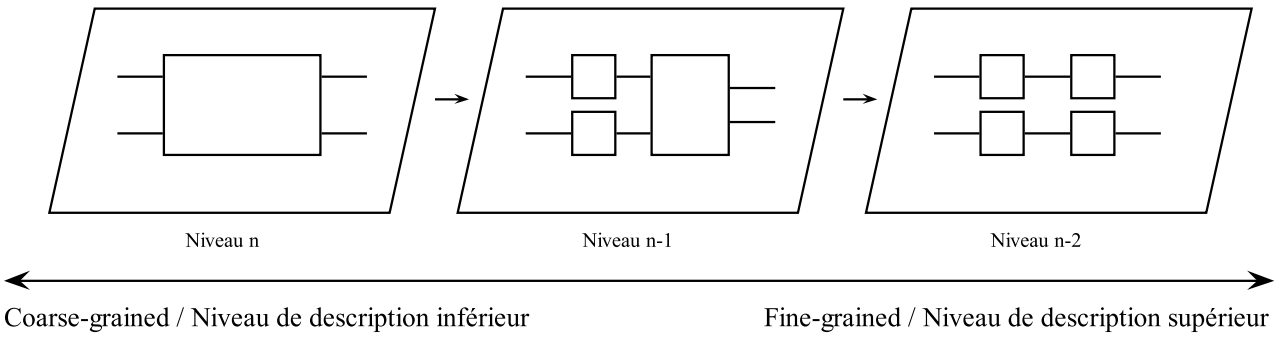


FIGURE 4.8 – La notion de granularité dans les niveaux de description.

La figure 4.5 représente un modèle ayant un niveau de description supérieur à celui de la figure 4.4.

Au sein d'une vue, les niveaux de description sont liés. Dans la figure 4.9 le niveau de description nd1 ajoute plus de détails au niveau de description nd2. Ceci implique la présence du lien horizontal is_{+D} entre ces deux niveaux.

Essentiellement, la différence entre ce type d'abstraction, l'abstraction de description, et l'abstraction de réalisation se manifeste dans le fait que le passage d'un niveau de réalisation à un autre implique toujours un changement du point de vue impliqué, ce qui n'est pas le cas pour le passage d'un niveau de description à un autre. En plus, un niveau de réalisation supérieur permet à l'architecte logiciel d'aller directement dans une phase de développement plus avancée par rapport aux niveaux inférieurs. Cela se traduit par la définition de nouveaux concepts, plus proche du code source exécutable, dans les niveaux de réalisation

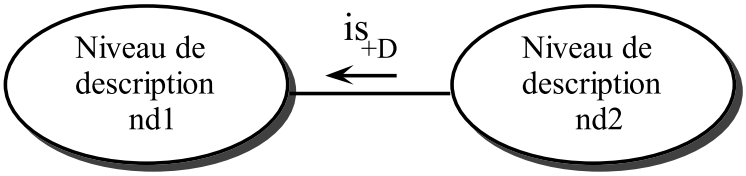


FIGURE 4.9 – Deux niveaux de description liés entre eux.

supérieurs. Cependant, un niveau de description supérieur ne permet pas à l'architecte logiciel de changer radicalement les concepts utilisés dans les niveaux inférieurs, mais il lui permet juste d'étendre les concepts utilisés auparavant dans les niveaux inférieurs, afin d'améliorer la clarté de ces modèles.

4.6.2 L'architecture organisée en une hiérarchie à trois niveaux

Une hiérarchie en général est une organisation d'éléments de façon que chacun d'entre eux soit subordonné à un autre. Cette notion a été abordée très souvent en informatique et surtout dans les domaines de structures de données. Actuellement nous sommes en train d'essayer d'intégrer cette notion dans la définition d'une architecture logicielle dans le cadre de notre approche *MoVAL*. Ainsi, on définit une hiérarchie comme étant une organisation de différents types d'éléments architecturaux (qui sont les vues et les niveaux d'abstraction) afin d'avoir une description architecturale hiérarchisée décrivant les exigences du système en considération, d'une manière suffisamment claire et non-ambigue.

Une architecture *MoVAL* est une hiérarchie à trois niveaux, classés du plus haut au plus bas :

1. les vues : les vues architecturales représentent la tête de la pyramide de la hiérarchie d'une architecture logicielle *MoVAL* ;

2. les niveaux de réalisation : chaque vue d'une architecture logicielle dans MoVAL est composée d'un ensemble de niveaux de réalisation liés entre eux par des liens marqués "is_{+A}", afin d'assister l'architecte logiciel à construire et développer cette vue tout le long du processus de définition de l'architecture ;

3. les niveaux de description : au sein d'un niveau de réalisation les niveaux de description sont liés entre eux par des liens marqués "is_{+D}". Ce niveau sert à donner à l'architecte logiciel le pouvoir et les outils de décrire, lors d'une phase particulière du processus de définition de l'architecture, un ensemble d'exigences et des préoccupations associées à un point de vue donné suivant plusieurs niveaux de granularité.

Normalement, un niveau de hiérarchie de description est pratiquement le conteneur de modèles dans une architecture logicielle, puisqu'il est le dernier niveau de hiérarchie défini dans l'approche *MoVAL*.
En se basant sur la définition des trois niveaux de hiérarchie proposés dans l'approche *MoVAL* et présentés au-dessus, une vue d'une architecture *MoVAL* peut être illustrée sous forme d'un graphe conceptuel orienté présentant les deux types de niveaux de hiérarchie de cette vue. La figure 4.10 représente un exemple d'un tel graphe.

Dans le graphe de la figure 4.10, les noeuds représentent les niveaux de description de l'architecture, liés par des arcs marqués "is_{+D}" pour indiquer que le noeud de destination représente un niveau de description supérieur au niveau de description représenté par le noeud source. Aussi, les niveaux de réalisation sont représentés par des rectangles pointillés et sont liés par des arcs marqués "is_{+A}" indiquant que le niveau de réalisation de destination est supérieur à celui de la source.
Pour aider l'architecte à visualiser la hiérarchie d'une architecture logicielle *MoVAL*, nous présentons dans la section 4.7 plusieurs types de diagrammes illustrant les différents éléments de cette hiérarchie et les liens qui peuvent avoir lieu entre eux.

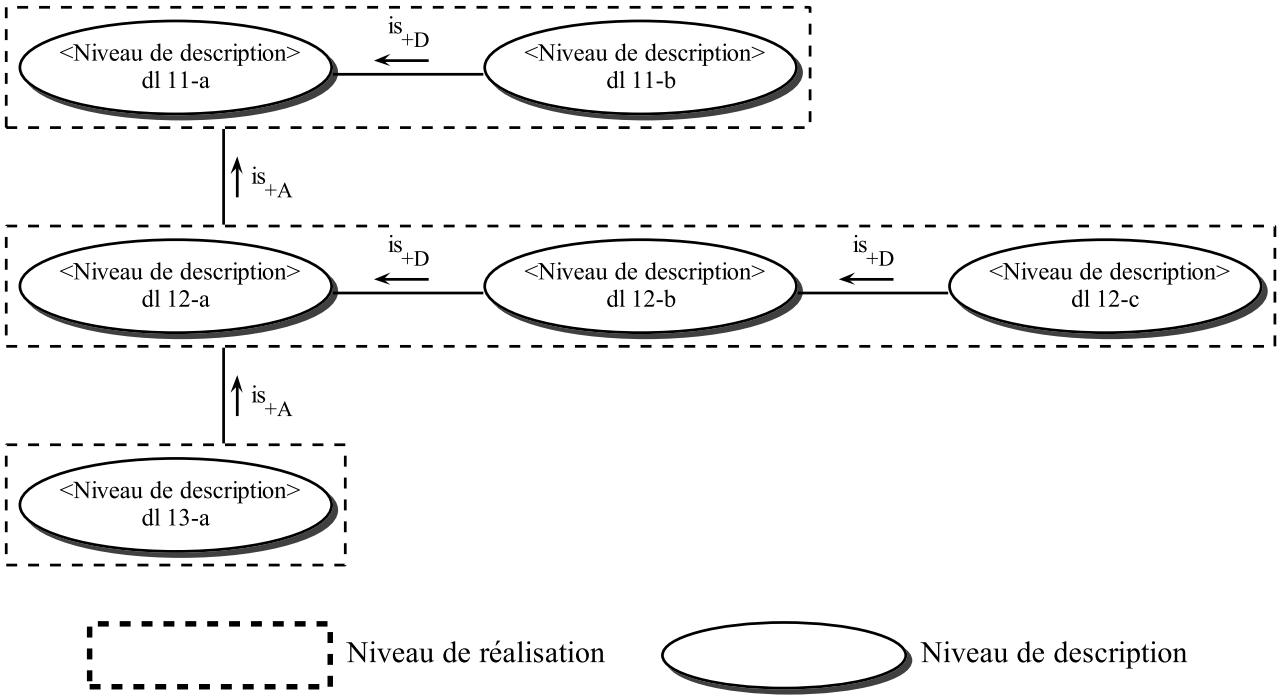


FIGURE 4.10 – Un graphe conceptuel orienté représentant une vue d'une architecture MoVAL.

La correspondance entre les niveaux de réalisation

En réalité, l'un des traits les plus importants de l'approche *MoVAL* est que toutes les vues de l'architecture peuvent être incorporées ensemble pour aboutir à une architecture logicielle rigoureuse et consistante d'une manière simple et efficace.
Cette incorporation peut être effectuée par l'architecte logiciel en définissant des correspondances entre les niveaux de réalisation des vues distinctes de l'architecture, à partir de liens marqués "$is_{=}$". En effet, il peut relier des niveaux de réalisation appartenant à des vues distinctes de l'architecture en déclarant que ces derniers représentent en fait la même phase dans le processus de définition de l'architecture.
Par ailleurs, pour que l'architecte puisse visualiser les liens de correspondance d'une manière simple et agréable, nous avons créé un type de diagramme appelé matrice de correspondance qui sera présenté dans la section 4.7.

4.7 Les diagrammes de visualisation d'une architecture *MoVAL*

Afin d'assister au mieux l'architecte lors de son travail, nous avons pensé d'ajouter de nouveaux diagrammes qui lui permettent de visualiser la structure de son architecture selon le type et le niveau d'information souhaité. Il peut opter soit pour **la représentation hiérarchique détaillée** ou pour **la représentation linéaire**. S'il souhaite visualiser les correspondances entre les différents niveaux de réalisation des vues de l'architecture, il peut utiliser **la matrice de correspondance**.

4.7.1 La représentation hiérarchique détaillée

Dans la représentation hiérarchique détaillée, l'architecte logiciel peut visualiser les quatre niveaux de la hiérarchie de l'architecture. Au premier niveau se trouvent les vues ; au second niveau les niveaux de réalisation ; au troisième niveau les niveaux de description ; et au quatrième niveau les modèles. Ainsi, la figure 4.11 illustre une visualisation dérivée du graphe conceptuel de la figure 4.10 présentée d'une manière plus aisée pour l'architecte logiciel.

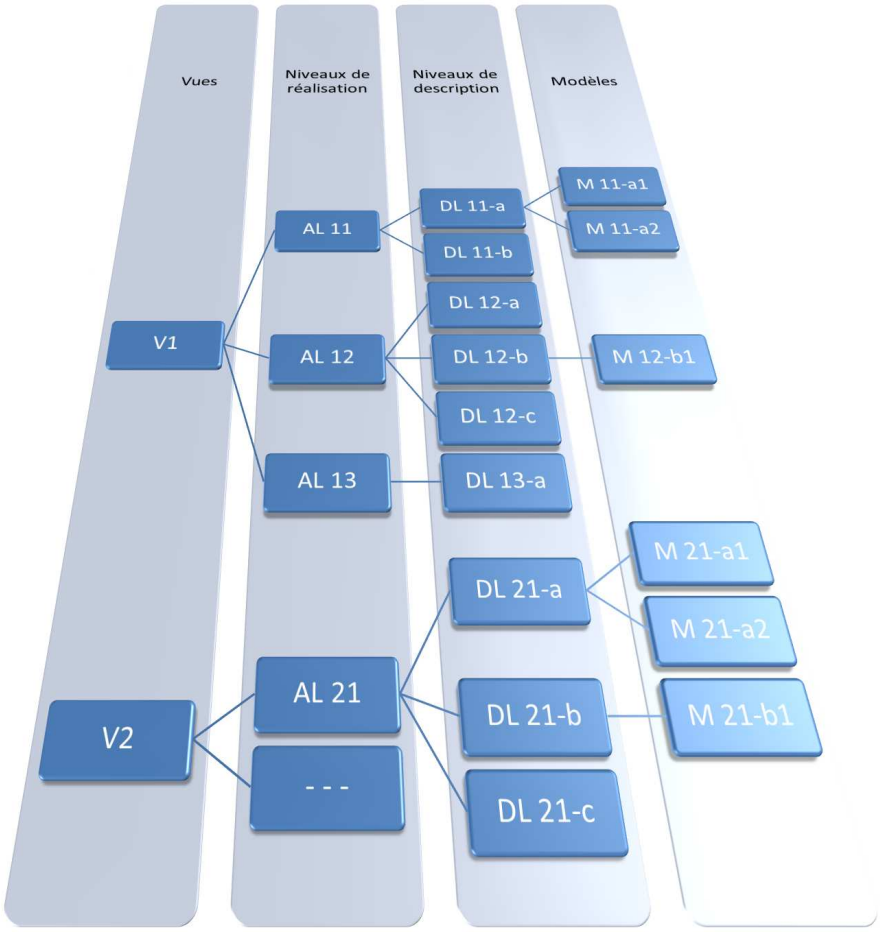


FIGURE 4.11 – La représentation hiérarchique d'une architecture logicielle.

4.7.2 La représentation linéaire

La représentation linéaire est une représentation de haut niveau de l'architecture logicielle *MoVAL* plus abstraite que la représentation hiérarchique détaillée. Seules les vues et les niveaux de réalisation sont représentés. La figure 4.12 illustre un exemple d'une représentation linéaire d'architecture.
Dans cette représentation une vue est représentée par un axe orienté sur lequel sont placés les niveaux de réalisation du haut (niveau le plus abstrait) vers le bas (niveau le plus concret). À noter qu'un niveau de réalisation est représenté par un point noir situé sur l'axe.

4.7.3 La représentation matricielle de correspondance

La représentation matricielle de correspondance est obtenue en partant d'une représentation linéaire et en ajoutant ensuite les relations de correspondance entre les différents niveaux de réalisation des vues de l'architecture. Ainsi, une relation de correspondance entre deux niveaux de réalisation de deux vues, représentée par une ligne droite pointillée joignant ces deux derniers, est créée pour identifier l'égalité de niveau de réalisation entre les deux différents niveaux. Cette relation est libellée "$is\_$". la figure 4.13 illustre la représentation matricielle induite de la figure 4.12.
Cette représentation est dénotée matricielle puisqu'elle sera ordonnée suite à la définition des relations de correspondance pour qu'elle sera comme illustrée dans la figure 4.14. Un label peut être ajouté sous la ligne de la relation indiquant la nature de cette relation qui peut être soit "*arch-defined*" ou "*derived*".
La matrice de correspondance peut représenter aussi des liens de consistance par une lettre entre deux crochets indiquant la sémantique du lien, et cela dans une nouvelle couche. La lettre peut être soit *R* pour indiquer un lien de référence, *D* pour la dépendance ou *P* pour la prédominance. La figure 4.15 représente

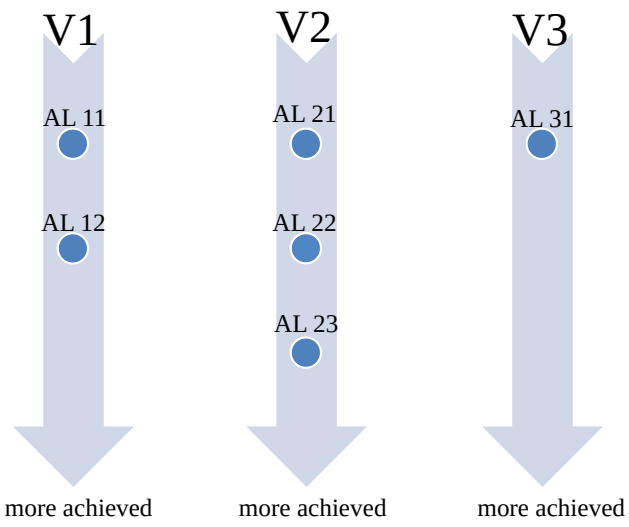


FIGURE 4.12 – La représentation linéaire d'une architecture logicielle.

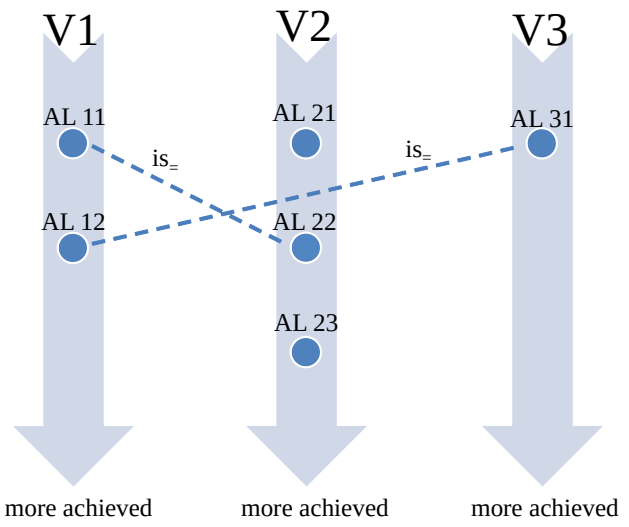


FIGURE 4.13 – La représentation matricielle de correspondance d'une architecture logicielle.

la matrice obtenue après l'addition des liens de consistance.
À noter que l'ordre de la relation de correspondance n'est pas total entre les niveaux de réalisation d'une architecture logicielle dans l'approche *MoVAL*, puisqu'on peut toujours trouver des niveaux incomparables. Pourtant, dans des cas idéaux, les relations de correspondance définies aboutissent à une correspondance totale sur l'architecture logicielle s'ils sont suffisants.

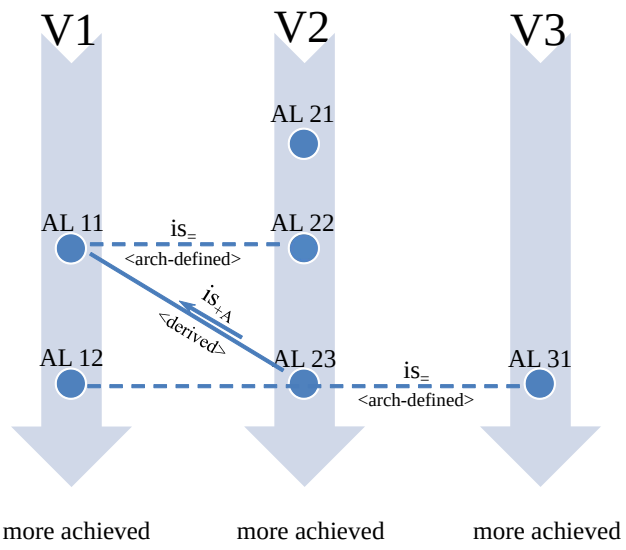


FIGURE 4.14 – La représentation matricielle de correspondance ordonnée d'une architecture logicielle.

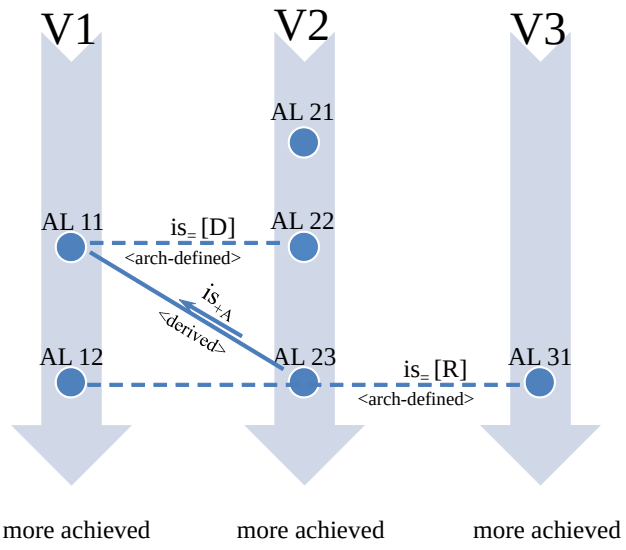


FIGURE 4.15 – La représentation matricielle de correspondance ordonnée et représentant les relations d'une architecture logicielle.

4.8 Les liens architecturaux dans *MoVAL*

Un lien architectural est une représentation formelle d'une relation qui existe entre deux éléments d'une architecture logicielle. Ces éléments peuvent être soit des niveaux de réalisation, des niveaux de description

ou des modèles. Un lien est défini dans l'un des deux contextes suivants :

1. le contexte **Intra-vue** : dans ce contexte le lien dénote une relation entre deux éléments d'une même vue ;
2. le contexte **Inter-vues** : dans ce cas le lien dénote une relation entre deux éléments appartenant à deux vues distinctes de l'architecture logicielle.

4.8.1 Les types de liens architecturaux

Comme déjà vu, nous avons identifié plusieurs types de liens structurels entre les éléments d'une architecture *MoVAL* :

- le lien d'appartenance $\in$:
  ce lien défini entre deux éléments architecturaux, exprime le fait qu'un élément architectural fait partie ou appartient à un autre élément de l'architecture. Ce type de lien est toujours défini implicitement par l'architecte logiciel (*i.e. arch-defined*) lors de la définition de la hiérarchie de l'architecture et ne peut pas avoir des attributs sémantiques spécifiques ;
- le lien de contenance $\supset$:
  c'est le lien inverse du lien appartenance ($\in$) ;
- le lien "is_{+A}" :
  exprime qu'un niveau de réalisation destination répond mieux à la question "Comment ?", ainsi il est plus concret (*i.e.* plus proche de l'implémentation) que le niveau de réalisation source. Ce lien peut être soit défini par l'architecte logiciel (*i.e. arch-defined*) ou dérivé à partir d'autres liens (*i.e. derived*) selon des règles qui seront présentées plus loin dans la section 4.8.2 ;
- le lien "is_{+D}" :
  exprime qu'un niveau de description effectue un *Zoom in* sur le niveau source. Ce lien peut être défini directement par l'architecte logiciel (*i.e. arch-defined*) ou dérivé (*i.e. derived*) à partir d'autres liens comme présenté dans la section 4.8.2 ;
- le lien "is_R" :
  ce lien est défini entre deux modèles architecturaux distincts. Il exprime qu'un modèle est relié (*i.e. is related*) à un autre modèle. Les sémantiques qui peuvent être soutenues et exprimées par ce lien sont la composition/agrégation, l'expansion/compression et la concrétisation/abstraction ;
- le lien de correspondance "$is_=$" :
  exprime une relation de correspondance (voir la section 4.6.2) entre deux niveaux de réalisation distincts appartenant à deux vues distinctes de l'architecture logicielle. Ce lien est soit défini par l'architecte (*i.e. arch-defined*) ou dérivé (*i.e. derived*) selon la règle qui sera présentée plus loin dans la section 4.8.2.

4.8.2 Les règles de dérivation des liens architecturaux

Les règles de dérivation du lien is_{+A}

- **RDER1 :** La dérivation par combinaison
  Un lien is_{+A} peut être dérivé par une combinaison d'un lien is_{+A} avec un lien de correspondance $is_=$. En effet, si un niveau de réalisation *al1* est lié à un autre niveau de réalisation d'une autre vue *al2* par un lien de correspondance $is_=$, et en même temps le niveau *al2* est lié à un autre niveau *al3* par un lien is_{+A}. Alors, on déduit par combinaison que le niveau *al1* est lié au niveau *al3* par un lien dérivé is_{+A}. La figure 4.16 donne un exemple sur la dérivation du lien is_{+A} par combinaison.

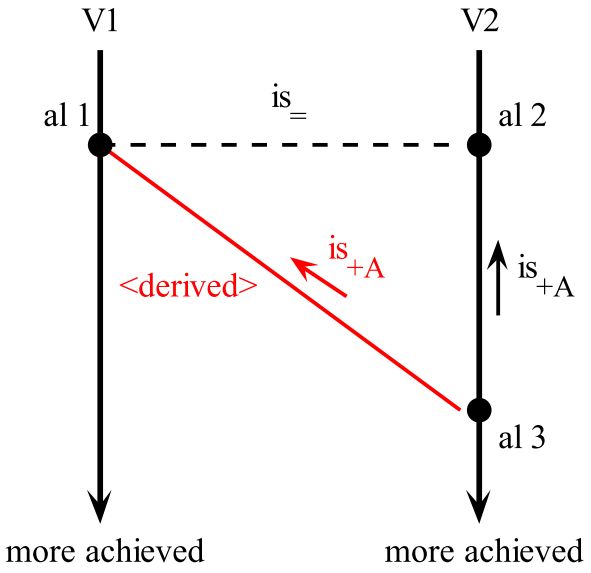


FIGURE 4.16 – La dérivation du lien *is+A* par combinaison des liens is_{+A} et $is_{=}$.

- **RDER2 :** La dérivation par transitivité
  Un lien is_{+A} peut être dérivé par transition. En effet, si un niveau de réalisation *al1* est lié à un autre niveau de réalisation *al2* par un lien is_{+A}, et en même temps le niveau *al2* est lié à un autre niveau de réalisation *al3* par un lien is_{+A}. Alors, on déduit par transitivité que le niveau *al1* est lié au niveau *al3* par un lien dérivé is_{+A}. La figure 4.17 donne un exemple sur la transitivité des liens is_{+A}.

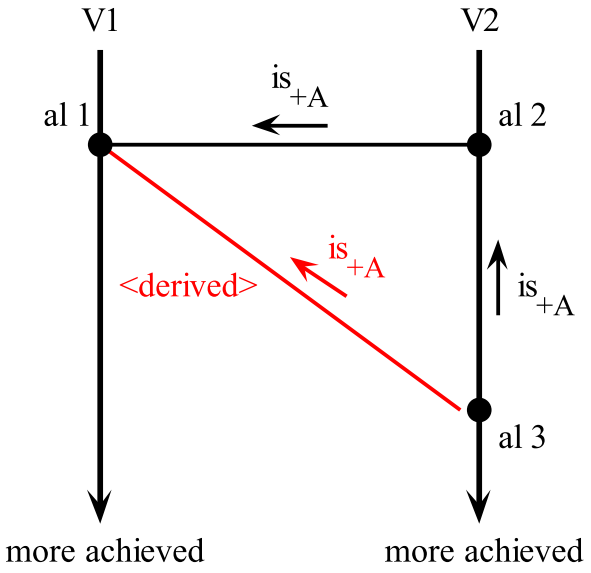


FIGURE 4.17 – La transitivité du lien is_{+A}.

La règle de dérivation du lien is_{+D}

- **RDER3 :** La dérivation par transitivité
  Un lien is_{+D} peut être dérivé par transition. En effet, si un niveau de description *dl1* est lié à un autre niveau de description *dl2* par un lien is_{+D}, et en même temps le niveau *dl2* est lié à un autre niveau de description *dl3* par un lien is_{+D}. Alors, on déduit par transitivité que le niveau *dl1* est lié au niveau *dl3* par un lien dérivé is_{+D}. La figure 4.18 donne un exemple sur la transitivité des liens is_{+D}.

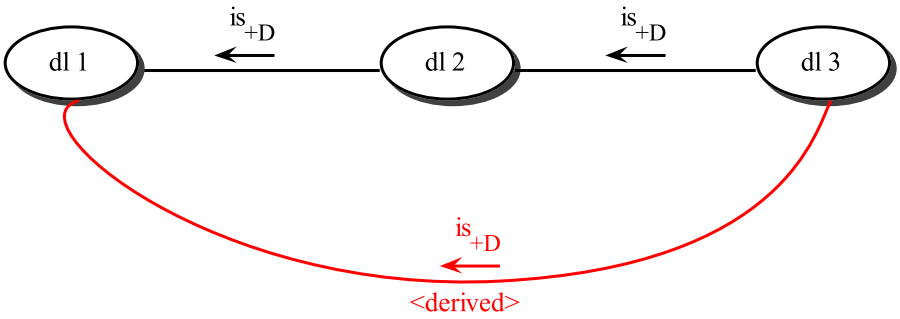


FIGURE 4.18 – La transitivité du lien is_{+D}.

La règle de dérivation du lien $is_{=}$

- **RDER4 :** La dérivation par transitivité
  De même, un lien $is_{=}$ peut être dérivé par transitivité. Alors, si un niveau de réalisation *al1* est lié à un autre niveau de réalisation *al2* par un lien $is_{=}$, et en même temps le niveau *al2* est lié à un autre niveau de réalisation *al3* par un lien $is_{=}$. Alors, on déduit par transitivité que le niveau *al1* est lié au niveau *al3* par un lien dérivé $is_{=}$. La figure 4.19 donne un exemple sur la transitivité des liens $is_{=}$.

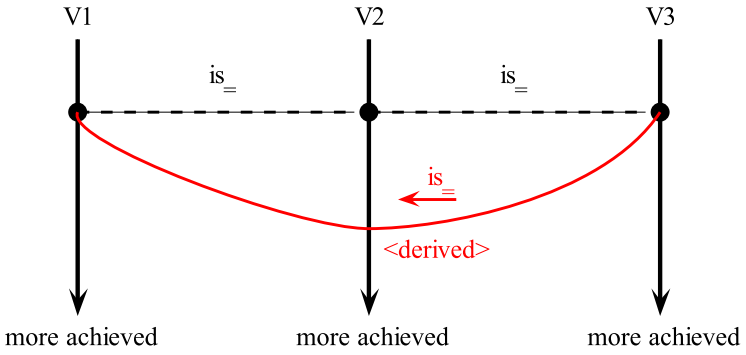


FIGURE 4.19 – La transitivité du lien de correspondance $is_{=}$.

La règle de dérivation du lien is_{R}

- **RDER5 :** La dérivation par transitivité
  De même, un lien is_{R} peut être dérivé par transitivité. Alors, si un modèle *m1* est lié à un autre modèle *m2* par un lien is_{R}, et en même temps le modèle *m2* est lié à un autre modèle *m3* par un lien is_{R}. Alors, on déduit par transitivité que le modèle *m1* est lié au modèle *m3* par un lien dérivé is_{R}.

4.8.3 La sémantique associée aux liens architecturaux

La sémantique associée aux liens architecturaux est définie par le biais de labels sémantiques apposés sur les liens structurels. Ces labels donnent au lien des sémantiques additionnelles qui peuvent être utilisées ultérieurement pour comprendre mieux l'architecture logicielle ou même pour générer du code source. On peut les classer par type de lien. Ainsi, on a associé les labels sémantiques suivants pour les liens $is_{=}$, is_{+A} et is_{+D} :

- **Dépendance/Prédominance** : pour déclarer que l'existence du niveau destination *dépend* du niveau source, ou que le niveau source *prédomine* le niveau destination. La définition de ce type de liens implique que la suppression ou la modification du niveau source rend nécessaire la suppression ou la modification respectivement du niveau destination ;

- **Référence** : le niveau de réalisation source réfère au niveau destination pour qu'il soit suffisamment clair, compréhensible, et complet. Ceci implique que la suppression ou la modification du niveau destination doit être suivie d'une révision du niveau source ;

Pour le lien is_R qui tient entre des modèles :

- **Concrétisation/Abstraction** : dans laquelle l'architecte logiciel déclare l'existence d'une *concrétisation* (*resp. abstraction*) explicite d'un élément du modèle source par un autre élément du modèle destination, c.à.d. un rapprochement de l'implémentation en changeant le formalisme utilisé. À noter que dans le cas de ce label sémantique, les niveaux de réalisation associés aux modèles source et destination sont strictement distincts appartenant à la même vue ;

- **Composition/Agrégation** : dans laquelle l'architecte logiciel déclare l'existence d'une *composition* (*resp. agrégation*) d'éléments d'un modèle du niveau de description source dans un autre modèle du niveau de description destination ;

- **Expansion/Compression** : dans laquelle l'architecte définit une *expansion* (*resp. compression*) d'un élément donné dans un modèle du niveau de description source, en un autre élément plus (resp. moins) détaillé (*i.e.* en utilisant plus (resp. moins) de traits du formalisme) dans un autre modèle du niveau de description destination ;

- **Connexion** : dans laquelle l'architecte logiciel définit une simple connexion entre deux modèles d'un niveau de description donné, tout en définissant les règles de consistance ou les contraintes nécessaires.

4.8.4 Tableau récapitulatif des différents types de liens et de leurs caractéristiques

Le tableau 4.2 résume les différents types de liens architecturaux ainsi que leurs propriétés principales. Ces propriétés sont :

- **Contexte** : spécifiant si le lien peut avoir lieu au sein d'une seule vue (*i.e.* intra-vue), ou entre deux vues différentes (*i.e.* inter-vues) ;

- **Genre** : identifiant si le lien peut être considéré également depuis ses deux extrémités, donc il sera bidirectionnel. Sinon, il sera unidirectionnel.

- **Notation** : définissant la notation utilisée pour illustrer l'existance du lien ;

- **Type de l'élément source/destination** : spécifiant le type potentiel de l'élément source/destination de ce lien ;

- **Nature** : spécifiant si le lien est défini par l'architecte logiciel (*i.e. User-defined*) ou dérivé (*i.e. Derived*) à partir d'autres liens ;

- **Label sémantique** : donnant au lien des sémantiques additionnelles qui peuvent être utilisées ultérieurement pour organiser les modèles entre eux ou même pour générer du code source (voir section 4.8.3) ;

- **Règles de dérivation** : identifiant les règles de dérivation permettant de déduire le lien à partir d'autres liens.

TABLE 4.2 – Les caractéristiques des différents types de liens dans MoVAL.

	$is_{=}$	is_{+A}	is_{+D}	is_{R}	Appartenance	Contenance
Contexte	Inter-vues	Intra-vue Inter-vues	Intra-vue	Intra-vue	Intra-vue	Intra-vue
Genre	Bidirectionnel	Unidirectionnel	Unidirectionnel	Unidirectionnel	Unidirectionnel	Unidirectionnel
Notation	$is_{=}$	is_{+A}	is_{+D}		✖	✖
Type élément source	Niveau de réalisation	Niveau de réalisation	Niveau de description	Modèle	Niveau de réalisation Niveau de description	Niveau de description Modèle
Type élément destination	Niveau de réalisation	Niveau de réalisation	Niveau de description	Modèle	Niveau de description Modèle	Niveau de réalisation Niveau de description
Nature	arch-defined derived	arch-defined derived	arch-defined derived	arch-defined	arch-defined	arch-defined
Label sémantique	Référence Dépendance/ Prédominance	Référence Dépendance/ Prédominance	Référence Dépendance/ Prédominance	Concrétisation/Abstraction Composition/Agrégation Expansion/Compression Connexion	✖	✖
Règles de dérivation	RDER4	RDER1, RDER2	RDER3	RDER5	✖	✖

4.9 La cohérence de l'architecture MoVAL

L'organisation hiérarchique de l'architecture logicielle lui confère un ensemble d'avantages qui ne peuvent pas être ignorés, comme la simplification de la complexité de l'architecture, l'adaptabilité de l'architecture avec les différents types et niveaux de compréhension, etc.
Cependant, la description de l'architecture logicielle ne sera complète sans la définition de règles assurant sa cohérence. Pour cela, dans cette partie nous nous intéressons à cette fonctionnalité indispensable et complémentaire à toute architecture multi-vues.
Dans ce qui suit, nous considérons la cohérence sous deux aspects : **la cohérence structurelle** et **la cohérence sémantique**.

4.9.1 La cohérence structurelle

Ce type de cohérence assure qu'une architecture *MoVAL* "soit bien formée" ou que sa structure soit valide. Cette cohérence est assurée par un ensemble de règles portant sur une seule vue (*i.e.* règles intra-vue) et d'autres portant sur plusieurs vues (*i.e.* règles inter-vues).

- Les règles intra-vue sont :
  - **RINTRA1 :** dans une architecture logicielle il existe au moins une vue ;
  - **RINTRA2 :** dans une vue, tous les niveaux de réalisation sont comparables entre eux par la relation is_{+A} ;
  - **RINTRA3 :** au sein d'un niveau de réalisation, tous les niveaux de description sont comparables entre eux par la relation is_{+D} ;
  - **RINTRA4 :** le lien is_{+D} ne peut pas être défini entre des niveaux de description contenus dans des niveaux de réalisation distincts.
- Les règles inter-vues sont :
  - **RINTER1 :** soient deux vues *v1* et *v2*, un niveau de réalisation *al1* de *v1* est au plus lié par un lien $is_{=}$ à un autre niveau de réalisation de *v2* ;
  - **RINTER2 :** le lien is_{+A} entre deux niveaux de réalisation de deux vues est toujours *derived* (*i.e.* ne peut pas être *arch-defined*) ;

– **RINTER3 :** les liens $is_{=}$ et is_{+A} sont mutuellement incompatibles, c.à.d. on ne peut pas avoir deux niveaux de réalisation *al1* et *al2*, appartenant respectivement aux vues *v1* et *v2*, liés par les liens $is_{=}$ et is_{+A} simultanément.

Exemple d'une architecture non bien formée

Soit l'architecture de la figure 4.20 composée de deux vues *V1* et *V2* telles que :

- *V1* contient deux niveaux de réalisation *al1* et *al3*/ *al3* is_{+A} *al1*
- *V2* contient deux niveaux de réalisation *al4* et *al2*/ *al2* is_{+A} *al4*
- de plus on a *al1* $is_{=}$ *al2* et *al3* $is_{=}$ *al4*

Cette architecture n'est pas bien formée car elle viole la règle RINTER3, en effet on aura :

1. *al1* is_{+A} *al4* (par la règle de dérivation RDER1).
   $\begin{cases} al1\ is_{=}\ al2(parhypothse) \\ al2\ is_{+A}\ al4(parhypothse) \end{cases} \Rightarrow al1\ is_{+A}\ al4$
2. al3 is_{+A} al4 (par la règle de dérivation RDER2).
   $\begin{cases} al3\ is_{+A}\ al1(parhypothse) \\ al1\ is_{+A}\ al4(parhypothse) \end{cases} \Rightarrow al3\ is_{+A}\ al4$
3. $\begin{cases} al3\ is_{+A}\ al4(parhypothse) \\ al3\ is_{=}\ al4(parhypothse) \end{cases} \Rightarrow$ RINTER3 violée

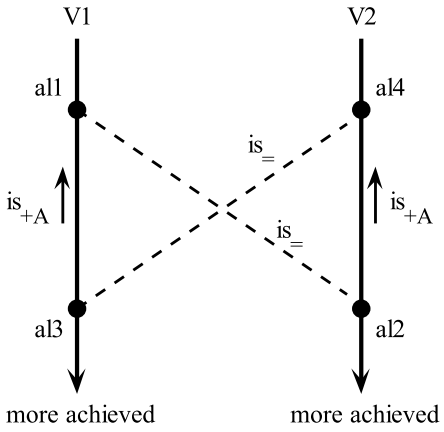


FIGURE 4.20 – Un cas violant la règle RINTER3.

4.9.2 La cohérence sémantique

La cohérence sémantique de l'architecture logicielle est assurée en respectant les règles suivantes :

- **RSEM1 :** Si un niveau de réalisation *al1* est lié à un autre niveau de réalisation *al2* par un lien de dépendance, alors la suppression ou la modification du niveau *al2* doit être suivie d'une suppression ou modification du niveau *al1* ;
- **RSEM2 :** Si un niveau de réalisation *al1* est lié à un autre niveau de réalisation *al2* par un lien de prédominance, alors la suppression ou la modification du niveau *al1* doit être suivie d'une suppression ou modification du niveau *al2* ;

- **RSEM3 :** Si un niveau de réalisation *al1* est lié à un autre niveau de réalisation *al2* par un lien de référence, alors la suppression ou la modification du niveau *al2* doit être suivie d'une révision du niveau *al1* ;
- **RSEM4 :** S'il existe un lien dérivé (*i.e. derived*) par transition entre deux éléments *e1* et *e3* d'une architecture, tel que *e1* lié à *e2* par *l1* et *e2* lié à *e3* par *l2*, alors tous les labels sémantiques communs entre *l1* et *l2* doivent être définis directement sur un lien *arch-defined* entre *e1* et *e3* ;
- **RSEM5 :** La suppression d'un élément de l'architecture *e1* implique la suppression de tout autre élément lié à *e1* par un lien d'appartenance.

4.10 Le méta-modèle de *MoVAL*

Tous les notions de base et les extensions définies dans l'approche *MoVAL* et représentées dans les sections de ce chapitre ont été modélisées et transformées en un méta-modèle selon le niveau M2 du standard *MOF*. La figure 4.21 représente le méta-modèle obtenu.

En effet, ce méta-modèle est conforme avec celui du standard IEEE 42010, et alors garde une partie de son méta-modèle et ajoute de nouvelles parties correspondantes à la contribution apportée par l'approche *MoVAL*. Plusieurs éléments de ce méta-modèle ont été hérités du méta-modèle IEEE comme :

- - Système,
- - Architecture,
- - Description d'architecture,
- - Intervenant,
- - Point de vue,
- - Vue,
- - Préoccupation,
- - Etc.

En même temps, d'autres éléments ont été ajoutés comme :

- - Niveau d'abstraction,
- - Niveau de réalisation,
- - Niveau de description,
- - Formalisme,
- - Lien,
- - Etc.

Un système (*System*), comme été défini dans le standard IEEE, n'est pas limité à des applications individuelles, mais il peut être aussi un sous-système, système de systèmes, et toutes autres agrégations logicielles. Un système possède toujours des intervenants (*Stakeholder*), qui participent dans les différentes phases de construction de ce système. Ces intervenants peuvent être des personnes, équipes, et même des organisations qui possèdent des intérêts dans le système, comme par exemple l'architecte logiciel, les développeurs, les analystes, les experts du domaine, les utilisateurs, etc.

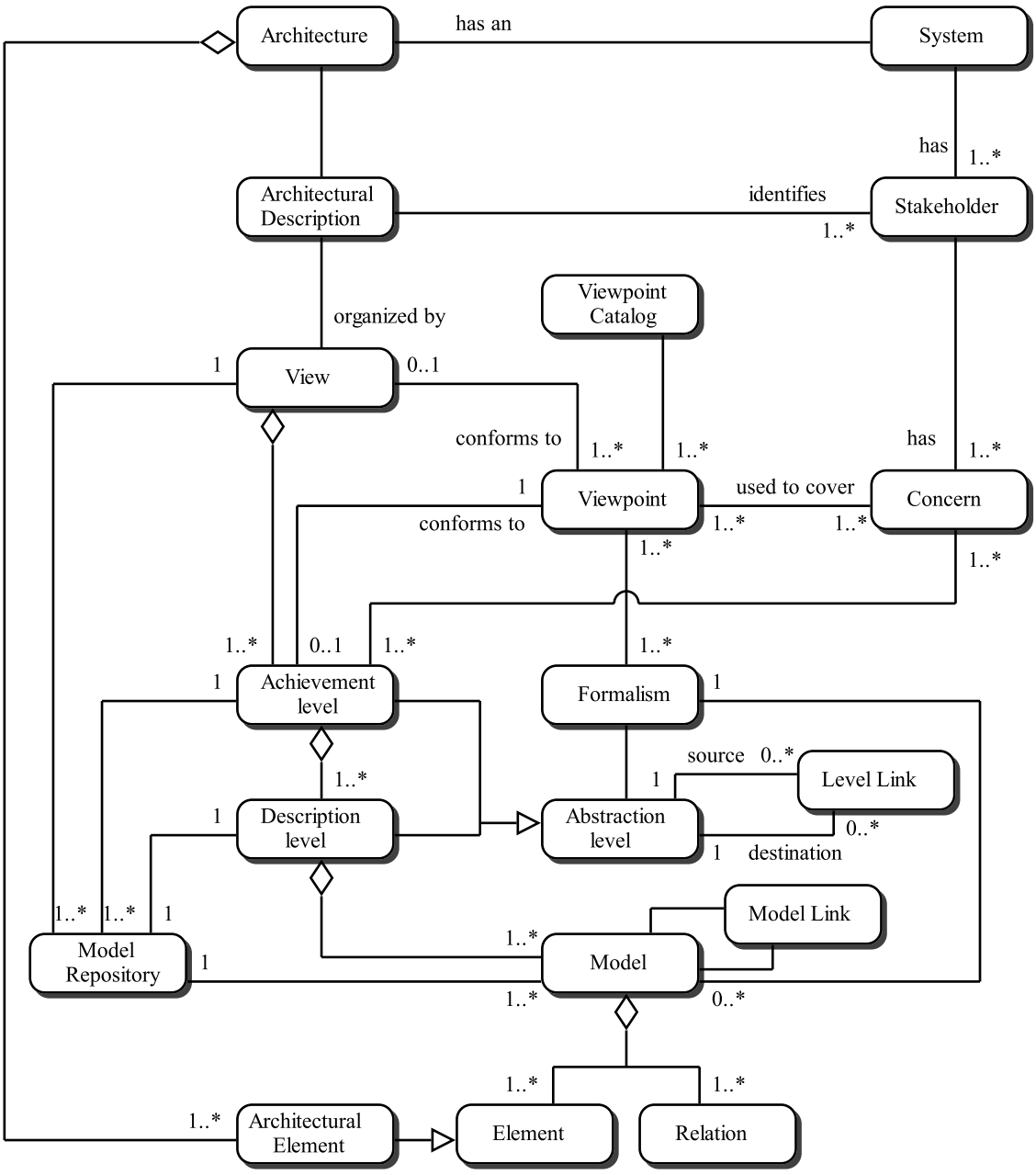


FIGURE 4.21 – Le méta-modèle de MoVAL.

En effet, chaque intervenant se focalise, selon ses intérêts, sur une partie bien spécifique des exigences du système, qui seront définies sous forme de préoccupations (*Concern*).
Simultanément, un système sera toujours associé à une architecture (*Architecture*) regroupant et organisant ses éléments (*Element*), qui représentent, selon la technologie adoptée, des composants, services, modules, objets, etc.
Une architecture sera documentée à travers une description d'architecture (*Architectural Description*), qui est composée d'une hiérarchie de vues (*View*) et de niveaux d'abstraction (*Abstraction Level*).
Les niveaux d'abstraction peuvent être de deux types, les niveaux de réalisation (*Achievement Level*) et les niveaux de description (*Description Level*), et ils peuvent être liés l'un à l'autre par des liens (*Link*).
Un niveau de réalisation d'une vue doit être conforme avec un point de vue (*Viewpoint*) qui lui définit les formalismes (*Formalism*) et les préoccupations générales qui doivent être respectées dans la construction de cette vue. En plus, chaque niveau de réalisation définit des préoccupations supplémentaires spécifiques à un système particulier.
Les niveaux de description sont composés de modèles (*Model*) qui sont construit via des concepts (*Concept*) qui peuvent être eux même des éléments de l'architecture. Ces niveaux de description possèdent en plus des répertoires de modèles (*Model Repository*) définissant l'adresse physique de leurs modèles.
Finalement, les points de vue peuvent être prédéfinis dans un catalogue (*Viewpoint Catalog*) pour qu'ils seront réutilisés dans tous les projets d'un domaine spécifique.

4.11 La configuration architecturale

Dans *MoVAL*, nous définissons la notion d'une configuration architecturale qui a pour but de minimiser et de simplifier la représentation de l'architecture logicielle considérant un sous-ensemble limité d'aspects qui coupent transversalement les vues de l'architecture, comme par exemple les aspects sécurité ou concurrence. Egalement, une configuration d'architecture peut être considérée pour représenter, en un niveau d'abstraction donné, les préoccupations d'une catégorie d'intervenants exprimées déjà dans une ou plusieurs vues de l'architecture.
En effet, une configuration architecturale est un sous-ensemble de l'ensemble des modèles de l'architecture, choisis pour représenter les aspects voulus ou la perspective des intervenants adressés. Ainsi, une configuration architecturale est définie par l'ensemble des intervenants pour lesquels elle est adressée, et par l'ensemble des répertoires contenant les modèles choisis. Cette configuration architecturale peut être visualisée par les diagrammes présentés dans la section 4.7 de ce chapitre.

4.12 Un catalogue de points de vue dans MoVAL

Afin de préparer le terrain pour une application aisée et efficace de l'approche *MoVAL*, nous allons présenter dans cette section une configuration possible du méta-modèle *MoVAL*, ou un catalogue de point de vue, regroupant leurs préoccupations et leurs formalismes. Normalement, cette configuration est carrément indépendante de l'approche *MoVAL* elle-même. Ainsi, l'approche *MoVAL* peut être adoptée séparément du catalogue selon les domaines d'application des systèmes voulus.
Les points de vue de ce catalogue et leurs préoccupations et formalismes vont être présentés selon le patron de documentation des points de vue proposé en annexe dans le standard *IEEE 42010* [ISO/IEC/IEEE, 2011].
Ce catalogue contient les points de vue : Contexte (*Context*), Capacités Fonctionnelles (*Functional Capabilities*), Implémentation Fonctionnelle (*Functional Implementation*), Information (*Information*), Données Logiques (*Logical Data*), Données Physiques (*Physical Data*), et Déploiement (*Deployment*).

4.12.1 Le point de vue Contexte

Nom : Contexte, en anglais *Context*.
Idée générale : Décrire les relations, dépendances, et interactions entre le système et son environnement.
Préoccupations :

- Identifier les utilisateurs du système.
- Identifier les entités physiques avec lesquelles le système interagit.
- Identifier les ressources matérielles à la disposition du système.
- Définir les systèmes, services, ou n'importe quelles entités logicielles externes avec lesquelles le système interagit, et définir le flux de données entre eux.
- Identifier les risques critiques dans lesquels le système peut tomber.

Intervenants : Tous les intervenants du système, et surtout les acquéreurs, analystes, utilisateurs, et développeurs.
Formalismes :

- Le formalisme textuel (i.e. rédaction d'un rapport).
- Diagramme de contexte en utilisant une notation dite "Boxes-and-lines". Ce diagramme place le système clairement dans son environnement et relit le avec les entités externes avec lesquelles il interagit. Il doit résumer les rôles et les responsabilités des intervenants contribuant dans ces interactions. Alors ce diagramme présente une vue globale du système dans son environnement et inclut typiquement les trois éléments suivants : le système, les entités externes, et les interfaces entre les deux premiers éléments.

4.12.2 Le point de vue Capacités Fonctionnelles

Nom : Capacités Fonctionnelles, en anglais *Functional Capabilities*
Idée générale : Décrire les capacités fonctionnelles du système pendant le temps d'exécution.
Préoccupations :

- Définir le style architectural approprié.
- Définir les capacités fonctionnelles du système.
- Définir les interfaces externes offertes par le système.
- Identifier les différentes applications et éléments logiciels de haut niveau du système.

Intervenants : Tous les intervenants du système.
Formalismes :

- Le formalisme textuel (i.e. rédaction d'un rapport).
- Diagramme UML de cas d'utilisation (i.e. Use case diagram).
- Diagramme UML d'état-transition.
- Diagramme représentant les différentes applications et éléments logiciels de haut niveau en utilisant une notation "Boxes-and-lines".

4.12.3 Le point de vue Implémentation Fonctionnelle

Nom : Implémentation Fonctionnelle, en anglais *Functional Implementation*
Idée générale : Décrire les éléments fonctionnels du système pendant le temps d'exécution, leurs responsabilités, interfaces, et interactions primaires.
Préoccupations :

- Définir la structure interne du système en termes d'éléments logiciels.
- Définir les interactions entre les éléments du système.

Intervenants : Les développeurs.
Formalismes :

- Diagramme UML de classes.
- Diagramme UML de composants.
- Diagramme UML d'activités
- Diagramme UML de séquence.
- Diagramme UML d'état-transition.

4.12.4 Le point de vue Information

Nom : Information, en anglais *Information*
Idée générale : Décrire les informations que le système doit sauvegarder.
Préoccupations :

- Définir les informations que le système doit sauvegarder.
- Assurer la complétude des informations à sauvegarder.
- Assurer l'intégrité des informations à sauvegarder.

Intervenants : Les acquéreurs, analystes, les utilisateurs, et l'administrateur des bases de données.
Formalismes :

- Le formalisme textuel (i.e. rédaction d'un rapport).
- Diagramme d'entité-relation.

4.12.5 Le point de vue Données Logiques

Nom : Données Logiques, en anglais *Logical Data*
Idée générale : Décrire la structure logique des bases de données.
Préoccupations :

- Définir la structure globale des bases de données du système.
- Définir les interactions entre les différentes bases de données du système.
- Définir la structure interne détaillée de chaque base de données du système.
- Minimiser la redondance des données.

- Maximiser l’intégrité des données.
- Améliorer la qualité et la consistance des données.
- Assurer que les structures logiques des bases de données du système sont tous conformes au 3NF.
- Déterminer le flux de données entre les différentes entités logicielles.

Intervenants : Les développeurs de base de données, et l’administrateur des bases de données.
Formalismes :

- Diagramme d’entité-relation.
- Diagramme illustrant la distribution des données en utilisant une notation "Boxes-and-lines".
- Diagramme de flux de données.

4.12.6 Le point de vue Données Physique

Nom : Données Physique, en anglais *Physical Data*
Idée générale : Décrire la structure Physique des bases de données.
Préoccupations :

- Construire les bases de données physiques.
- Déterminer la possession des données.
- Déterminer les privilèges sur les données.
- Définir les règles d’archivage des données.
- Assurer des scénarios de restauration de données.
- Assurer la possibilité de détecter les sources des modifications sur les données.

Intervenants : Les développeurs de base de données, et l’administrateur des bases de données.
Formalismes :

- Shéma physique des bases de données.

4.12.7 Le point de vue Déploiement

Nom : Déploiement, en anglais *Deployment*
Idée générale : Décrire l’environnement dans lequel le système doit être déployé, et les dépendances que le système peut avoir sur ses éléments.
Préoccupations :

- Identifier les plateformes et les librairies dont le système a besoin pour être exécuté.
- Identifier le matériel nécessaire pour que le système soit efficace.
- Identifier les besoins en termes de systèmes partenaires.
- Identifier les contraintes de compatibilité avec les différentes technologies.
- Identifier les ressources réseaux nécessaires.

Intervenants : Les administrateurs de systèmes, les développeurs, les testeurs, les équipes de support.
Formalismes :

- Diagramme UML de déploiement.

4.13 Conclusion

Dans ce chapitre, nous présentons une approche baptisée *MoVAL* (*Model, View and Abstraction Level based software architecture*) [Kheir et al., 2013]. Cette approche a pour but d'offrir à l'architecte logiciel, des outils lui facilitant la tâche de construire une architecture qui soit documentée à travers une description architecturale compréhensible et facilement manipulable par différents intervenants.
En effet, cette approche répond aux motivations présentées dans le chapitre précédent. Elle se base autour de la construction d'architectures logicielles multipoints de vue et multi-granularité, à travers la définition de plusieurs vues pour une architecture logicielle donnée, puis la décomposition de chacune d'entre elle en des niveaux d'abstraction : les niveaux de réalisation et les niveaux de description.
En effet, cette approche est conforme au standard *IEEE 42010* [ISO/IEC/IEEE, 2011] qui a été conçu au sein de la communauté *APG* (*IEEE Architecture Planning Group*) afin de standardiser la définition des architectures logicielles multipoints de vue et leurs éléments cruciaux. Par conséquence, l'approche *MoVAL* hérite les définitions de ces éléments principaux qui font partie du standards *IEEE 42010*, comme les définitions d'une architecture logicielle, d'un point de vue, d'une vue, d'un modèle, etc. et les étend par les notions de niveau de réalisation, niveau de description, et lien.
Normalement, l'adoption de ce standard dans *MoVAL* avait pour but premièrement d'être conforme à un standard très répandu dans le monde industriel des architectures logicielles. Deuxièmement, cette adoption nous permet de construire notre approche sur la base d'un travail robuste de haut niveau sans avoir besoin de commencer à partir de zéro dans la conception d'une telle approche. D'une autre part, l'extension du standard *IEEE 42010* dans *MoVAL* avait pour but d'intégrer la notion de hiérarchie, qui n'a pas été considérée dans ce standard, au sein de l'approche *MoVAL* afin de réduire les complexités de développement des systèmes logicielles.
En plus, l'approche *MoVAL* adopte la structure logicielle définie dans le *MOF* (*Meta-Object Facility*) [OMG, 2013] par le groupe *OMG* (*Object Management Group*). Ce standard est basé sur l'ingénierie dirigée par les modèles, et alors il permet à l'architecte logiciel de construire ses architectures sur plusieurs niveaux parmi ceux définis dans le *MOF*, et surtout les niveaux *M1* et *M0*. Normalement, l'architecte logiciel peut organiser la construction et le développement du système logiciel et il peut gérer les relations qui peuvent exister entre les différents modèles de l'architecture quand il se localise dans le niveau *M1*. Pourtant, il peut organiser le déploiement et la maintenance d'un système logiciel et il peut gérer les relations entre les différents exécutables de ce système quand il se localise au niveau *M0*.

5

Formalisation de MoVAL

5.1 Introduction

La logique propositionnelle et la logique des prédicats sont utilisées depuis longtemps en informatique comme une investigation rationnelle de la notion de vérité dans les systèmes informatiques. Ainsi, afin de vérifier d'une part l'efficacité, la robustesse, et la fiabilité de l'approche présentée auparavant, et afin d'ouvrir, d'autre part, une porte sur la faisabilité de la vérification mathématique des architectures logicielles construites selon MoVAL, nous présentons dans cette partie une formalisation des concepts de base de cette approche dans la logique des prédicats.

5.2 Préambule

Avant de donner les définitions formelles des notions apportées dans *MoVAL*, nous considérons les ensembles et définitions suivantes :

- $\mathcal{SA}$: l'ensemble des architectures logicielles, ou *Software Architectures* ;
- $\mathcal{ST}$: l'ensemble des intervenants ou *stakeholders* ;
- $\mathcal{P}$: l'ensemble de noms des points de vue ;
- $\mathcal{V}$: l'ensemble des vues ;
- $\mathcal{O}$: l'ensemble d'objets typés ;
- $\mathcal{AL}$: l'ensemble des niveaux de réalisation ou *Achievement levels* ;
- $\mathcal{DL}$: l'ensemble des niveaux de description ou *Description levels* ;
- $\mathcal{M}$: l'ensemble des modèles ;
- $\mathcal{AC}$: l'ensemble des configurations possibles des architectures logicielles, ou *Architecture Configuration* ;
- $\mathcal{E}$: l'ensemble des éléments des modèles ;

- $\mathcal{R}$: l'ensemble des relations entre des éléments ;
- $\mathcal{FN}$: l'ensemble des fonctionnalités offertes par les éléments ;
- $\mathcal{SE}$: l'ensemble des sémantiques ou des préoccupations définies dans l'architecture ;
- $\mathcal{F}$: l'ensemble des formalismes utilisés dans le domaine ;
- $\mathcal{REP}$: l'ensemble des répertoire.

Definition 5.2.0.1. Le type d'un objet $o \in \mathcal{O}$ peut être obtenu en appliquant une fonction notée *t*,

$$t \ : \ \mathcal{O} \ \longrightarrow \ \mathcal{SA} \cup \mathcal{P} \cup \mathcal{V} \cup \mathcal{AL} \cup \mathcal{DL} \cup \mathcal{M}$$

Definition 5.2.0.2. L'identifiant d'un objet $o \in \mathcal{O}$ peut être obtenu en appliquant une fonction injective notée *i*,

$$i \ : \ \mathcal{O} \ \longrightarrow \ \mathcal{S}$$

$\mathcal{S}$: le domaine du type chaine de caractères.

5.3 Éléments, modèles, et relations entre éléments

Ayant défini les différents ensembles de la formalisation, nous introduisons maintenant les définitions formelles des différents éléments de l'approche. Premièrement, les définitions d'un élément d'un modèle et du modèle lui-même seront présentées. En plus, plusieurs types de relations entre les différents éléments et modèles de l'architecture seront définis.

5.3.1 Éléments

Definition 5.3.1.1. Un élément $e \in \mathcal{E}$ est défini par le tuple :

$$e = \langle n, t, FN \rangle$$

- *n* : l'identifiant de *e* ; *n* = *i*(*e*) ;
- *t* : le type de l'élément. Par exemple : classe, composant, entité, etc. ;
- FN : l'ensemble des fonctionnalités décrivant l'élément.

5.3.2 Relation entre les éléments

Quatre types de relations peuvent avoir lieu entre les éléments des modèles : la composition, l'agrégation, l'expansion et la compression.
Comme la composition définit des sémantiques inverses aux sémantiques définies par l'agrégation, et l'expansion définit des sémantiques inverses aux sémantiques définies par la compression, nous allons fournir juste les définitions de la composition et l'expansion.

Definition 5.3.2.1. La composition est une relation entre les éléments de deux modèles différents tel que : $\forall e_1, e_2 et\ e_3 \in \mathcal{E}$, e_1 *is_composed_of* e_2 si et seulement si :

a. e_1 est composite ;

a. e_2 est composant de e_1.

En effet cette relation est :

- non-réflexive : e_1 *is_composed_of* e_1 est toujours faux, puisque e_1 n'est pas composant de e_1 (régle (b) violée) ;
- non-symétrique : e_1 *is_composed_of* e_2 n'implique pas e_2 *is_composed_of* e_1, puisque si e_2 est composant de e_1, alors e_1 n'est pas composant de e_2 (régle (b) violée) ;
- transitive : si e_1 *is_composed_of* e_2 et e_2 *is_composed_of* e_3, alors e_1 *is_composed_of* e_3, puisque :
  - e_1 est composite,
  - comme e_2 est composant de e_1 et e_3 est composant de e_2, alors e_3 est composant de e_1.

  D'où e_1 *is_composed_of* e_3.

Definition 5.3.2.2. L'expansion est une relation entre les éléments de deux modèles différents tel que : $\forall e_1, e_2 et\ e_3 \in \mathcal{E}$, e_1 *is_expansion_of* e_2 si et seulement si :

a. n(e_1) = n(e_2) ;

b. t(e_1) = t(e_2) ;

c. F(e_1) $\supset$ F(e_2) (l'ensemble des fonctionnalités décrivant e_1 contient l'ensemble des fonctionnalités décrivant e_2).

En effet, cette relation est :

- non-réflexive : e_1 *is_expansion_of* e_1 est toujours faux, puisque F(e_1) $\not\supset$ F(e_1) ;
- non-symétrique : e_1 *is_expansion_of* e_2 n'implique pas e_2 *is_expansion_of* e_1, puisque si F(e_1) $\supset$ F(e_2) alors F(e_2) $\not\supset$ F(e_1) ;
- transitive : si e_1 *is_expansion_of* e_2 et e_2 *is_expansion_of* e_3, alors e_1 *is_expansion_of* e_3, puisque :
  - si n(e_1) = n(e_2) et n(e_2) = n(e_3), alors n(e_1) = n(e_3),
  - si t(e_1) = t(e_2) et t(e_2) = t(e_3), alors t(e_1) = t(e_3),
  - si F(e_1) $\supset$ F(e_2) et F(e_2) $\supset$ F(e_3), alors F(e_1) $\supset$ F(e_3).

  D'où e_1 *is_expansion_of* e_3.

Definition 5.3.2.3. La concrétisation est une relation entre les éléments de deux modèles différents tel que : $\forall e_1, e_2 \in \mathcal{E}$, e_1 *is_concreteness_of* e_2 si et seulement si :

a. n(e_1) = n(e_2) ;

b. t(e_1) $\neq$ t(e_2) ;

c. F(e_1) $\supset$ F(e_2) (l'ensemble des fonctionnalités décrivant e_1 contient l'ensemble des fonctionnalités décrivant e_2).

En effet, cette relation est :

- non-réflexive : e_1 *is_concreteness_of* e_1 est toujours faux, puisque t(e_1) = t(e_1) (règle (b) violée) ;
- non-symétrique : e_1 *is_concreteness_of* e_2 n'implique pas e_2 *is_concreteness_of* e_1, puisque si F(e_1) $\supset$ F(e_2) alors F(e_2) $\not\supset$ F(e_1) (règle (c) violée) ;
- non-transitive : si e_1 *is_concreteness_of* e_2 et e_2 *is_concreteness_of* e_3, n'implique pas que e_1 *is_concreteness_of* e_3, puisque si t(e_1) $\neq$ t(e_2) et t(e_2) $\neq$ t(e_3), on peut avoir t(e_1) = t(e_3) (règle (c) violée).

5.3.3 Modèles

Definition 5.3.3.1. Un modèle $m \in \mathcal{M}$ est défini par le tuple :

$$m = \langle n, E, R \rangle$$

- *n* étant l'identifiant de *m* ; *n* = *i*(*m*) ;
- E $\subset \mathcal{E}$, est l'ensemble des éléments constituants du modèle *m* ;
- R $\subset \mathcal{R}$, est l'ensemble des relations entre les éléments de *m* ;
- *m* peut être défini par les ensembles de ses éléments et leurs relations.
  Ainsi, nous définissons $\bigoplus$ un opérateur tel que :
  $\bigoplus : \mathcal{E} \times \mathcal{R} \longrightarrow \mathcal{M}$
  tel que : *m* = E $\bigoplus$ R ;
- $\forall E' \subset E, \forall R' \subset R, E' \bigoplus$ R' est un sous-modèle de *m*.

Definition 5.3.3.2. Chaque modèle *m* définit des sémantiques spécifiques à lui à travers ses éléments et leurs relations. Ainsi, on définit une fonction *Sem* qui rend les sémantiques définies par un modèle *m*.

$$Sem \ : \ \mathcal{M} \ \longrightarrow \ \{\mathcal{S}\}$$

les propriétés suivantes sont toujours valides :

- *Sem*(*m*) = *Sem*(E $\bigoplus$ R) ;
- $\forall E' \subset E, \forall R' \subset$ R, *Sem*(E' $\bigoplus$ R') $\subset$ *Sem*(E $\bigoplus$ R).

5.3.4 Relations entre les modèles

Definition 5.3.4.1. Nous définissons une relation is_R entre les modèles, tel que :
$\forall m_1, m_2 \in \mathcal{M}, m_1 \ is_R \ m_2 \Leftrightarrow$
$\exists e_1 \in E(m_1), \exists E \subset E(m_2), \forall e_2 \in E, e_1$ is_composed_of e_2, or $\exists e_1 \in E(m_1), \exists e_2 \in E(m_2), e_1$ is_expansion_of e_2, or $\exists e_1 \in E(m_1), \exists e_2 \in E(m_2), e_1$ is_concreteness_of e_2.

Definition 5.3.4.2. Dans le cas de composition, une relation is_{+A} entre les modèles sera définie comme suit :
$\forall m_1, m_2 \in \mathcal{M}, m_1$ *is*$_{+A}$ m_2
$\Leftrightarrow \exists e_1 \in E(m_1), \exists E_2 \subset E(m_2), \exists R_2 \subset R(m_2)$
$/e_1 = E_2 \bigoplus R_2$
À noter que cette même définition peut être appliquée dans le cas d'une relation is_{+D}, en tenant compte que les éléments des modèles peuvent changer de nature selon le formalisme utilisé dans le cas d'une relation is_{+A}. Contrairement au cas des relations is_{+D}, dans lesquels les éléments des modèles conservent toujours leur nature.

Definition 5.3.4.3. Dans le cas de l'expansion, une relation is_{+A} entre les modèles sera définie comme suit :
$\forall m_1, m_2 \in \mathcal{M}, m_1$ *is*$_{+A}$ m_2
$\Leftrightarrow \exists E_1 \subset E(m_1), \exists R_1 \subset R(m_1), \exists E_2 \subset E(m_2), \exists R_2 \subset R(m_2)$
$/$*Sem*$(E_1 \bigoplus R_1) \subset$ *Sem*$(E_2 \bigoplus R_2)$

Definition 5.3.4.4. Dans le cadre de cette définition on va définir les deux prédicats suivants :

- Soit le prédicat *Change*(*e*) (*e* $\in (E)$) indiquant que l'élément *e* a été modifié ;

- Soit le prédicat *Valid*(*m*) ($m \in (M)$) indiquant que le modèle *m* est valide.

Pour la connexion, une relation is_{+A} entre les modèles sera définie comme suit :
$\forall m_1, m_2 \in \mathcal{M}, m_1\ is_{+A}\ m_2$
$\Leftrightarrow \exists e_1 \in E(m_1), \exists e_2 \in E(m_2), Change(e_1) \wedge Valid(m_2) \Rightarrow Change(e_2)$
De même, cette définition peut être appliquée dans le cas d'une relation is_{+D}.

5.4 Niveaux de description, niveaux de réalisation, et leurs relations

5.4.1 Niveaux de description

Definition 5.4.1.1. Un niveau de description dl $\in \mathcal{DL}$ est défini par le tuple :

$$dl = \langle n, M \rangle$$

- n étant l'identifiant de *dl* ; *n* = *i*(*dl*) ;
- M $\subset \mathcal{M}$, est l'ensemble des modèles associé à *dl*.

5.4.2 Niveaux de réalisation

Definition 5.4.2.1. Un niveau de réalisation $al \in \mathcal{AL}$ est défini par le tuple :

$$al = \langle n, DL, is_{+D}, p \rangle$$

- *n* étant l'identifiant du niveau de réalisation ; *n* = *i*(*al*) ;
- DL $\subset \mathcal{DL}$ est l'ensemble des niveaux de description associé à *al* ;
- is_{+D} est une relation d'ordre sur DL ;
- $p \in \mathcal{P}$ est le point de vue auquel al est conforme.

5.4.3 Relation inter-niveaux

Definition 5.4.3.1. La relation is_{+A} entre les niveaux de réalisation est une relation d'ordre totale définie sur l'ensemble des niveaux de réalisation d'une vue *v*, noté AL(*v*). Cette relation est non-réflexive, non-symétrique, et transitive. Elle est définie par l'architecte.
Cette relation vérifie la propriété suivante :
$\forall al_1, al_2 \in AL(v), \forall dl_i \in DL(al_1), dl_j \in DL(al_2)$
$al_2\ is_{+A}\ al_1 \Rightarrow \nexists m, n, m \in M(dl_i), n \in M(dl_j)/m\ is_{+A}\ n$

Definition 5.4.3.2. La relation is_{+D} est une relation d'ordre total définie sur l'ensemble des niveaux de description relatifs à un niveau de réalisation $al \in \mathcal{AL}$, noté DL(*al*). Cette relation est non-réflexive, non-symétrique, et transitive. Elle est définie par l'architecte.
$\forall al \in \mathcal{AL}, \forall dl_1, dl_2 \in DL(al)$,
$dl_2\ is_{+D}\ dl_1 \Rightarrow \nexists m \in M(dl_1), n \in M(dl_2)/m\ is_{+D}\ n$

5.5 Architecture logicielle, vue, répertoire et configuration d'architecture

5.5.1 Architecture logicielle

Definition 5.5.1.1. Une architecture logicielle $sa \in \mathcal{SA}$ est définie par le tuple :

$$\langle a, V, ST \rangle$$

où :

- *a* étant l'identifiant de *sa* ; *a* = *i*(*sa*) ;
- V $\subset \mathcal{V}$ est l'ensemble des vues associées à *sa* ;
- ST $\subset \mathcal{ST}$ est l'ensemble des intervenants concernés par *sa*..

Definition 5.5.1.2. Soit *ST* l'ensemble des intervenants concernés dans une architecture logicielle quelconque, tel que ST $\subset \mathcal{ST}$.
Il existe une application *get_stakeholders*, de *ST* dans *C* qui est toujours valide pour cette architecture, tel que :

$$\begin{array}{rccc} \textit{get\_stakeholders} & : & \mathcal{ST} & \longrightarrow & \mathcal{ST} \\ & & ST & \longmapsto & C \end{array}$$

C étant un sous-ensemble de $\mathcal{ST}$ représentant les classes d'intervenants qui doivent être considérées dans cette architecture.

5.5.2 Vue

Definition 5.5.2.1. Un point de vue $p \in \mathcal{P}$ est défini par le tuple :

$$p = \langle n, ST, SE, F \rangle$$

- *n* étant l'identifiant du point de vue ; *n* = *i*(*p*) ;
- $ST \subset \mathcal{ST}$: l'ensemble des intervenants dont le point de vue *p* représente leur vision vis à vis du système / *Card*(ST) ≥ 1 ;
- $SE \subset \mathcal{SE}$: l'ensemble des sémantiques et des préoccupations qui doivent être représentées à partir de *p* ;
- $F \subset \mathcal{F}$: l'ensemble des formalismes qui peuvent être utilisés pour exprimer les sémantiques de *p*.

Definition 5.5.2.2. Il existe une autre application *assign_viewpoint* de *C* dans $C \times P$, telque :

$$\begin{array}{rccc} \textit{assign\_viewpoint} & : & \mathcal{ST} & \longrightarrow & \mathcal{ST} \times \mathcal{P} \\ & & C & \longmapsto & C \times P \end{array}$$

P représente l'ensemble des points de vue qui doivent être considérés dans cette architecture.
Par conséquence, l'application *get_stakeholders*∘*assign_viewpoint* donne les points de vue d'une architecture quelconque en fonction des intervenants considérés.

Definition 5.5.2.3. Une vue $v \in \mathcal{V}$ est définie par le tuple :

$$v = \langle n, P, AL, is_{+A} \rangle$$

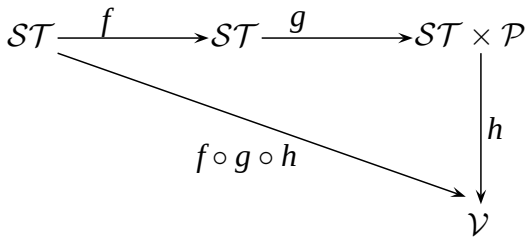
- *n* étant l'identifiant de la vue ; $n = i(v)$;
- P l'ensemble de points de vue dominants cette vue ;
- $AL \subset \mathcal{AL}$: un ensemble de niveaux de réalisation associé à *v* ;
- is_{+A} est la relation d'ordre totale sur AL définie dans 5.4.3.1.

Definition 5.5.2.4. Il existe une application *build_view* tel que :

$$\begin{array}{rccc} build\_view & : & \mathcal{ST} \times \mathcal{P} & \longrightarrow & \mathcal{V} \\ & & C \times P & \longmapsto & V \end{array}$$

V étant l'ensemble des vues candidates de l'architecture *a*.

Definition 5.5.2.5. En se basant sur les définitions 5.5.1.2, 5.5.2.2 et 5.5.2.4, on peut déduire qu'il existe une application *get_stakeholders*∘*assign_viewpoint*∘*build_view* qui donne pour une architecture $\langle a, \text{V}, \text{ST} \rangle$ l'ensemble des vues *V* de cette architecture à partir de l'ensemble des intervenants ST du système en consideration.



5.5.3 Répertoire

Definition 5.5.3.1. Un répertoire $r \in \mathcal{REP}$ est défini par le tuple :

$$r = \langle v, al, dl \rangle$$

- v $\in \mathcal{V}$;
- $al \in AL(v)$;
- $dl \in DL(al(v))$.

Definition 5.5.3.2. Nous définissons la relation *in* qui lie un modèle à son répertoire, tel que : $\forall m \in \mathcal{M}, \exists rep / in(m, rep) \Rightarrow m \in M(dl(rep))$

5.5.4 Le prédicat de correspondance entre les niveaux de réalisation

$is_{=}$ est un prédicat indiquant la correspondance entre les différents niveaux de réalisation appartenant aux différentes vues de l'architecture, et qui peut être défini par l'architecte.

Definition 5.5.4.1. Le prédicat de correspondance $is_{=}$ est défini sur les niveaux de réalisation issues de deux vues différentes d'une même architecture :

$$is_{=} : AL(v) \times AL(v) \longrightarrow bool$$

- Il est défini par l'architecte ;

- Soit une architecture logicielle *sa* $\in \mathcal{SA}$, les vues $v_1, v_2 \in$ V(*sa*), les niveaux de réalisation $al_1 \in AL(v_1) et al_2 \in AL(v_2)$.
  Supposons que al_1 *is*$_=$ al_2.
  Le prédicat de correspondance vérifie les propriétés suivantes :
  - Il est symétrique, tel que :
    al_1 *is*$_=$ $al_2 \Leftrightarrow al_2$ *is*$_=$ al_1
  - $\forall al_j, (al_j \text{ is}_{+A} al_1 \Rightarrow al_j \text{ is}_{+A} al_2) \wedge \forall al_k, (al_1 \text{ is}_{+A} al_k \Rightarrow al_2 \text{ is}_{+A} al_k)$

Comme les noms des objets niveaux de réalisation doivent être uniques dans l'architecture, l'opération de définition de relation de correspondance doit être suivie par une fonction de renommage.

Definition 5.5.4.2. La fonction de renommage des niveaux de réalisations est définie comme suit :

$$\begin{array}{rccc} \textit{Rename} & : & \mathcal{A} \times String & \longrightarrow & \mathcal{A} \\ & & a & \longmapsto & a_{renamed} \end{array}$$

5.5.5 Configuration d'architecture

Definition 5.5.5.1. Une configuration d'architecture *ca* $\in \mathcal{AC}$ est définie par le tuple :

$$ca = \langle V, REP, M, ST \rangle$$

telque :

- V $\subset \mathcal{V}$;
- $\forall r \in REP, v(r) \in V$;
- M $\subset \mathcal{M}$;
  $\forall m \in M, \forall r \in REP, m \in dl(r)$;
- ST $\subset \mathcal{ST}$; l'ensemble des intervenants (*i.e. stakeholders*) intéressés par la configuration d'architecture.

5.6 Conclusion

Dans ce chapitre, nous avons représenté les éléments et les concepts de base de notre approche *MoVAL*, en utilisant la logique des prédicats. Cette formalisation mathématique avait pour but d'assurer et de démontrer la cohérence de toutes les définitions des différents éléments de l'approche entre eux, et de leur complétude afin de détecter les cas de défaillance dans lesquelles cette approche peut tomber et de les régler. Egalement, afin de déterminer les points faibles de l'approche sur lesquelles il faut revenir et améliorer.

6

MoVAL-ADP : le processus de définition d'architecture adapté à MoVAL

6.1 Introduction

Dans les chapitres précédents nous avions souligné l'importance des architectures logicielles et mis en évidence leurs nombreux avantages ainsi que leurs impacts sur la qualité globale des systèmes informatiques produits, comme l'organisation du développement des systèmes informatiques, la réduction des complexités de ces systèmes et la couverture ultime des exigences des différents intervenants, etc. Ensuite nous avions présenté notre contribution dans ce domaine, l'approche *MoVAL*, qui est une approche d'architecture logicielle multipoints de vue et multi-hiérarchies.
L'objectif de ce chapitre est d'introduire le processus de définition d'architecture logicielle (*ADP*) spécifique à l'approche *MoVAL*.
Tout au début, nous présentons les processus de développement de logiciels desquels nous nous sommes inspirés : le processus de développement logiciel unifié (*Unified process*), le processus de définition d'architecture de *Rozanski et Woods*, ainsi que les méthodes dites agiles. Ensuite, nous détaillons le processus de définition d'une architecture logicielle dans *MoVAL*. Enfin, nous terminons par un exemple illustrant l'adoption de *MoVAL-ADP* dans le cadre du cas d'étude SWVE introduit dans le chapitre 4 (section 4.3).

6.2 Les processus de définition d'architectures logicielles

Dans ce qui suit, nous présentons trois approches qui nous semblent les plus importantes en définition de processus d'architectures logicielles qui sont : (1) le processus proposé par *Rozanski et Woods* [Rozanski and Woods, 201
(2) le processus unifié (*Unified Process*) [Booch and Rumbaugh, 1999], ainsi que (3) les méthodes dites agiles [Fowler and Highsmith, 2001].
La définition de l'architecture logicielle commence au début du cycle de vie d'un projet informatique là où les exigences et les besoins associés au système informatique n'ont pas été encore fixés. Le but étant de résoudre les complexités et les risques qui menacent le développement du système avant de passer aux étapes de conception, de modélisation et d'implémentation. La définition de l'architecture logicielle s'établit en effectuant des activités formant un processus appelé le processus de définition d'architecture, ou *Architecture Definition Process*, abrégé en *ADP*. Le ADP ne remplace pas le processus de développement de logiciel, ou *Software Development Process*, abrégé en *SDP*.

En effet, le *ADP* est complémentaire au *SDP* et devrait s'effectuer en parallèle avec ce dernier. Les deux s'échangeant les flux d'informations. Cet aspect sera étudié en 6.5.

6.2.1 Le processus de définition d'architecture de *Rozanski et Woods*

Selon *Rozanski et Woods* [Rozanski and Woods, 2011], à l'entrée du processus de définition d'une architecture logicielle d'un système, le cadre et le contexte de ce système doivent être établis. De plus, il faut définir les intervenants principaux, ainsi que leurs préoccupations fondamentales, connues en anglais sous le nom "*First-cut concerns*".
Ensuite, le processus de définition d'architecture logicielle, cherche à consolider les entrées des données (contexte et *first-cut concerns*) puis à identifier tous les scénarios possibles que le système doit offrir. Après, les styles architecturaux appropriés doivent être identifiés afin de produire une architecture logicielle candidate. Finalement, l'architecte logiciel doit explorer les options architecturales et choisir la plus convenable afin de l'évaluer avec les intervenants, et décider si elle est acceptable ou non.
La figure 6.1 représente, par un diagramme d'activités, le processus de définition d'architecture proposé par *Rozanski et Woods*.

6.2.2 Le processus de développement logiciel unifié (*Unified process*)

Le *Unified Software Development Process*, ou simplement *le Unified Process* (*UP*), est un processus générique de développement logiciel qui est itératif et incrémental. Il a été décrit pour la première fois dans le livre intitulé "*The Unified Software Development Process*" [Booch and Rumbaugh, 1999]. Ce processus possède plusieurs implémentations, la plus réputée et la plus populaire est celle produite par l'*IBM* (*The International Business Machines Corporation*) et connue sous le nom *Rational Unified Process* (*RUP*).
Ce processus divise le projet en quatre phases différentes :

- *Inception Phase* : c'est la phase la plus courte du processus de développement dans laquelle il faut établir un *Business Case*, et il faut spécifier le cadre et le contexte du système informatique. Puis, il faut résumer les cas d'utilisation et les risques les plus importants qui peuvent avoir un impact sur la structure globale du système.

- *Elaboration Phase* : dans cette phase la majorité des exigences et des besoins du système doivent être capturés et la structure globale de ce système doit être validée en implémentant une référence d'architecture exécutable, plus connue sous le nom "*Executable Architecture Baseline*".

- *Construction Phase* : c'est la phase la plus longue du processus de développement dans laquelle tout le reste du système doit être implémenté.

- *Transition Phase* : dans cette phase le système doit être déployé afin de recevoir les réactions (*i.e. feedbacks*) des utilisateurs et leurs avis du système afin de le raffiner.

La figure 6.2 illustre le niveau d'effort relatif à chaque discipline en fonction des différentes phases du processus de développement.
En effet, comme ce processus de développement est itératif et incrémental, chaque phase peut être divisée en plusieurs itérations. En plus, ce processus tient en compte de l'aspect architectural des systèmes informatiques et propose la livraison d'une architecture logicielle candidate à la fin de la phase *Inception*, et de valider cette architecture durant la phase *Elaboration*.

6.2.3 Les méthodes agiles

Les processus de développement les plus utilisés actuellement dans l'industrie de développement logiciel sont les processus de la génération *agile* ayant un pourcentage d'adoption de *56.5%* par rapport aux autres

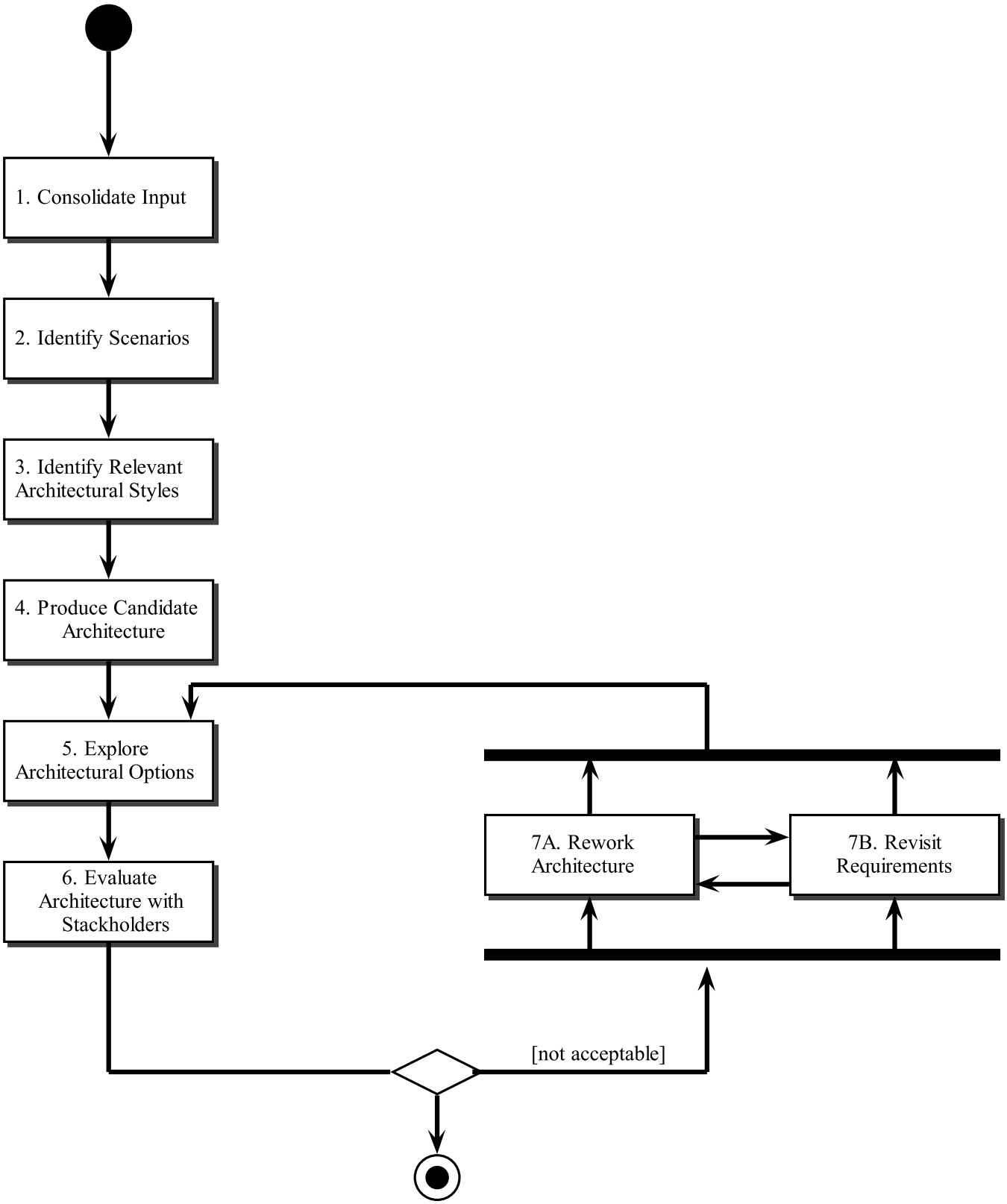


FIGURE 6.1 – Le processus de définition d'architecture proposé par *Rozanski et Woods*, extrait de [Rozanski and Woods, 2011].

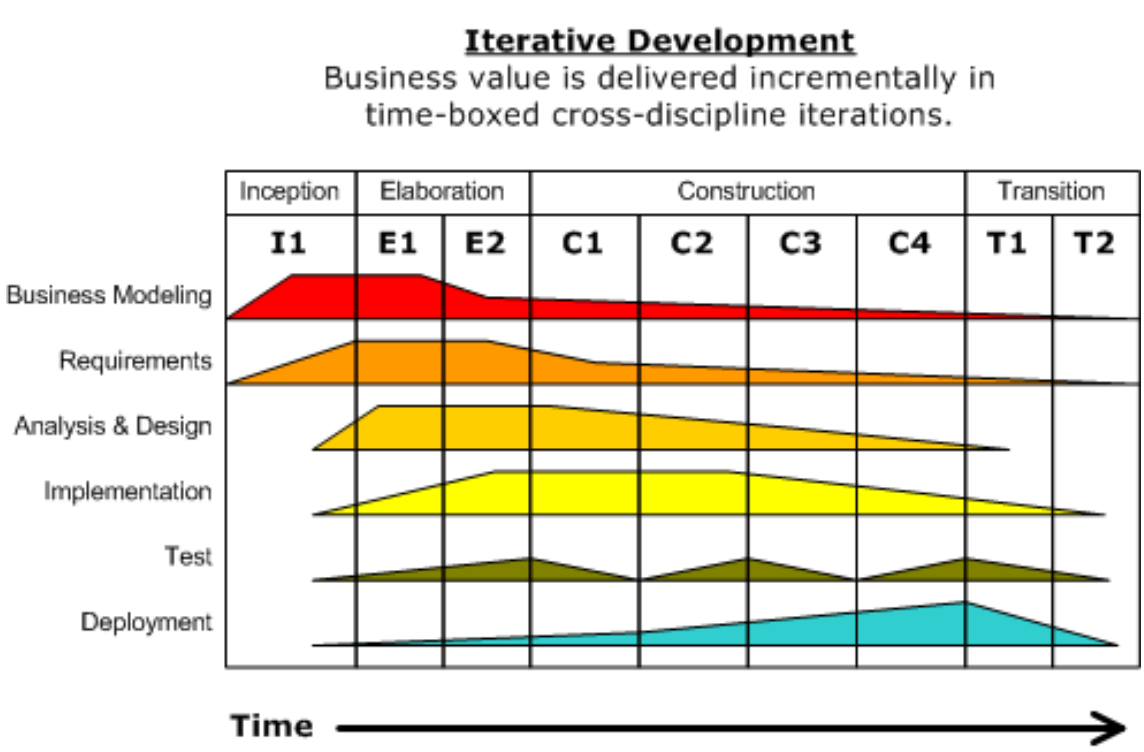


FIGURE 6.2 – Le niveau d'effort relatif à chaque discipline/Phase de développement.

processus selon le rapport établi en 2011 par *ExecutiveBrief* [ExecutiveBrief, 2011], un périodique dédié aux gestionnaires de la technologie et les chefs d'entreprise, et de *75%* selon le rapport [Vijayasarathy and Turk, 2008]. En plus, l'apport de cette génération de processus a été confirmé dans plusieurs autres rapports et études comme [Salo and Abrahamsson, 2008]. Cette génération de processus de développement regroupe des processus ayant les caractéristiques fondamentales suivantes : ce sont des processus itératifs et incrémentaux en premier lieu, en plus ils sont adaptatifs et soulignent un rythme de développement accéléré pour répondre aux besoins actuels de l'industrie informatique.
Parmi ces processus agiles, on peut mentionner : *Scrum*, *Extreme Programming* (*XP*) et *Rapid Application Development* (*RAD*).

Scrum

La méthode *Scrum* [Schwaber and Beedle, 2002] est une méthode qui vise à gérer un projet informatique afin qu'il soit délivré pendant une durée relativement réduite. Cette méthode s'appuie sur la segmentation et la division du projet informatique en plusieurs boîtes de temps, dites "*sprints*", qui peuvent avoir une durée entre des heures et un mois. Ainsi, le propriétaire du système informatique effectue une priorisation des exigences demandées afin que l'équipe de développement les organise dans des *sprints*. Normalement, chaque *sprint* commence par une vérification de la planification opérationnelle avant d'implémenter les exigences attribuées à ce *sprint*, et se termine par une démonstration de ce qui a été achevé. À la fin de tous les *sprints*, l'équipe aura un produit potentiellement livrable.
La figure 6.3 illustre le processus proposé dans la méthode *Scrum*.

Extreme Programming

La méthode *Extreme Programming* (*XP*) [Beck, 1999] est une méthode agile orientée particulièrement sur l'aspect réalisation d'un système informatique, contrairement à la méthode *Scrum* qui est orientée sur la gestion du projet informatique en général. Ainsi, cette méthode offre des pratiques de qualité de logiciel qui n'ont pas été offertes dans *Scrum*.

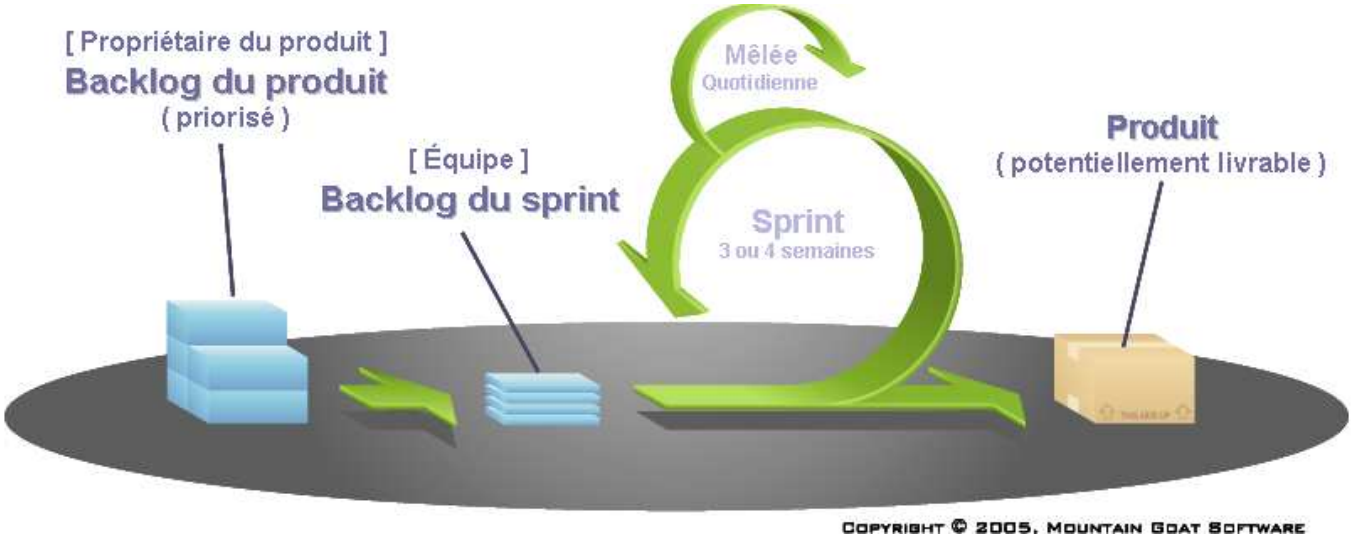


FIGURE 6.3 – Vue globale de la méthode *Scrum*, extraite de [Cohn, 2005].

Étant itérative, cette méthode s'appuie sur la division du processus de développement logiciel en plusieurs cycles ou itérations, de quelques semaines chacune. Chaque cycle consiste en plusieurs étapes :

- La spécification détaillée des besoins et des exigences client associées aux scénarios qui doivent être fournis à la fin de cette itération ;
- La transformation des scénarios en plusieurs tâches à réaliser et en des tests fonctionnels ;
- L'attribution de ces tâches aux développeurs disponibles pour les réaliser ;
- La réalisation des tâches ;
- La livraison du produit après que les tâches seront réalisées et les tests fonctionnels seront passés.

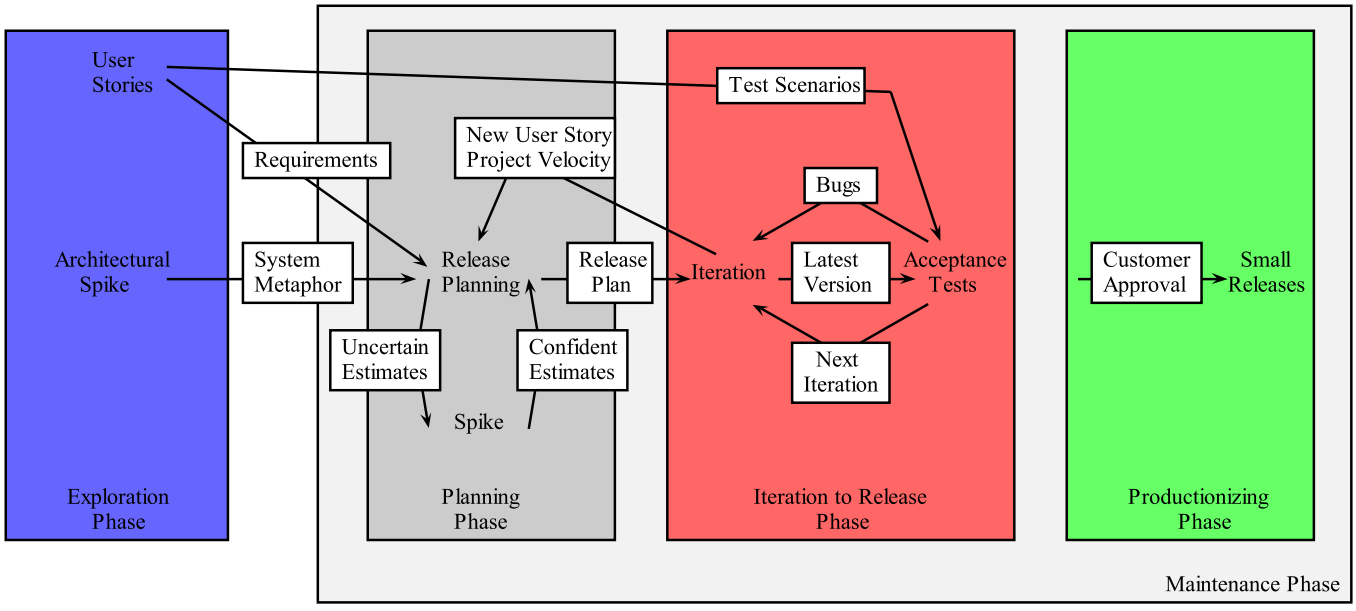


FIGURE 6.4 – Vue globale de la méthode *Extreme Programming*.

La figure 6.4 présente les quatre phases différentes de la méthodes *XP*. La phase "*Exploration*", qui consiste à dégager les besoins et les scénarios client qui vont être détaillés dans chaque itération, et consiste à

déterminer le style architectural qui va être utilisé. La deuxième phase, la phase "*Planning*", consiste à la division des scénarios acquis durant la première phase en plusieurs itérations. Durant la phase "*Iteration to release*" l'équipe de développement doit effectuer les étapes mentionnées ci-dessus pour chaque itération, avant d'effectuer les tests nécessaires pour que le produit de chaque itération soit accepté. La dernière phase, la phase "*Productionizing*" consiste à prendre l'approbation du client afin de fournir des petites livraisons.

Rapid Application Development

La méthode de développement rapide d'application (*Rapid Application Development, RAD*) est la première méthode de développement logiciel itératif, incrémental, et adaptatif. Donc elle peut être considérée la première méthode agile.
Cette méthode a été conçue comme un successeur du modèle "*Waterfall*" ajoutant la notion d'itération, et minimisant l'espace de planification et se concentrant plus sur les phases de conception et d'implémentation. Ce processus a été présenté et résumé par *James Martin* [Martin, 1991] par quatre phases fondamentales, comme illustré dans la figure 6.5 :

- La phase "*Requirements Planning*", dans laquelle une planification globale du système aura lieu, et une analyse des exigences et des préoccupations de ce système sera effectuée ;
- La phase "*User Design*". Durant cette phase, les utilisateurs interagissent avec les analystes et les développeurs du système informatique afin de développer des modèles et des prototypes représentant les besoins acquis ;
- La phase "*Construction*", dans laquelle les développeurs se concentrent sur l'implémentation du système, et la transformation des modèles construits dans la phase précédente en un code qui tourne. Durant cette phase aussi, les utilisateurs peuvent participer en fournissant leurs suggestions de changements ou améliorations en se basant sur les rapports et les *screenshots* produites par les développeurs ;
- Finalement la phase "*Cutover*". Cette phase ressemble à la phase transition du processus unifié, durant laquelle les derniers tests seront effectués, les données seront converties, et les utilisateurs seront engagés dans le système produit.

6.2.4 Positionnement de MoVAL-ADP par rapport aux approches étudiées

Même si le processus de définition d'architecture *MoVAL-ADP* est conforme avec le *Unified Process*, cela n'interdit pas qu'il peut aussi être adapté à d'autres processus de développement logiciels itératifs communs, comme les processus de développement de la génération agile. Cela revient au fait que les activités de ce processus ont été établies indépendamment de celles des processus de développement logiciel. Ainsi, le processus que nous proposons est générique et adaptable à n'importe quel processus de développement itératif, tout simplement en regroupant ses activités selon les phases prédéfinies dans le processus de développement adressé.

- Par rapport au processus unifié : *MoVAL-ADP* est complètement adapté au processus unifié (*UP*) en regroupant ses activités dans quatre phases fondamentales à l'instar de ce dernier. Ces quatre phases sont : (1) la phase *Inception*, (2) la phase *Elaboration*, (3) la phase *Construction*, et (4) la phase *Transition*. Ce qui lui ajoute tous les avantages du processus unifié et facilite aux équipes de développement son adoption grâce à leur familiarité avec le processus unifié et son application d'une manière vraiment efficace.
- Par rapport au processus proposé par *Rozansli et Woods* : De même, *MoVAL-ADP* est inspiré du processus proposé par *Rozanski et Woods*. En effet, il inclut de plus les activités visant l'organisation des modèles au sein des vues ainsi que la définition des liens entre les vues, etc. ...

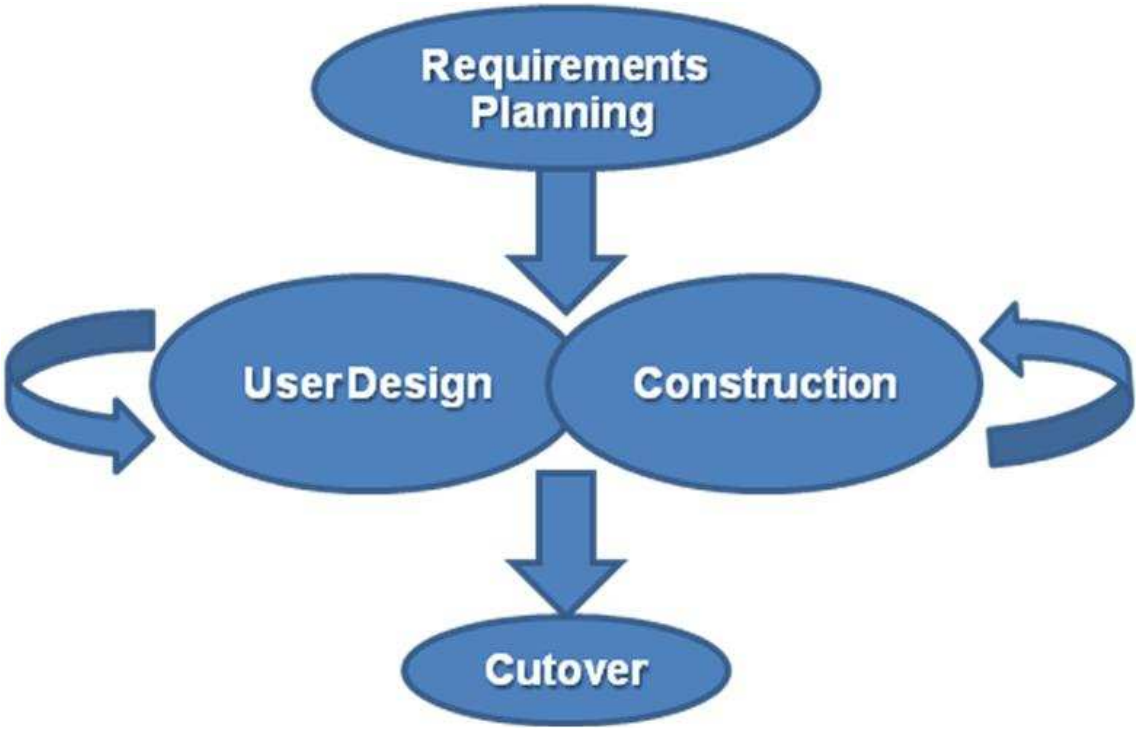


FIGURE 6.5 – Vue globale de la méthode *RAD*.

- Par rapport aux processus agiles : *MoVAL* se base sur l'implication ultime des intervenants dans le processus de définition de l'architecture logicielle. Ce processus vérifie les règles les plus importantes autour desquelles tous les processus agiles tournent (itératif, incrémental, évolutionnaire, etc.). Pour cela, *MoVAL-ADP* peut être facilement adapté à n'importe quel processus agile en distribuant les quatre différentes phases du processus *MoVAL-ADP* sur les phases du processus agile considéré.

6.3 Les principales caractéristiques de *MoVAL-ADP*

Le processus de définition d'architecture logicielle *MoVAL*, ou simplement *MoVAL-ADP*, se base sur un ensemble de propriétés et de caractéristiques fondamentales qui lui permettent d'être suffisamment efficace dans l'industrie de développement du logiciel. En effet, *MoVAL-ADP* est un :

- Processus incrémental et itératif :
  L'industrie informatique préfère actuellement adopter des processus de développement itératifs et incrémentaux comme le *Rational Unified Process* (*RUP*) et les méthodes de la génération agile, à l'instar de *Scrum* et *Extreme Programming* (*XP*). Afin qu'il soit en harmonie avec les processus de développement actuels, chaque phase du processus *MoVAL-ADP* peut être éventuellement parcourue en plusieurs itérations en ajoutant à l'issue de chaque itération un incrément à l'ensemble des artefacts de sortie demandés à la fin d'une phase.

- Processus adaptable avec les processus de développement incrémentaux :
  Les activités de définition de l'architecture sont regroupées dans des phases incarnant la méthodologie proposée dans les processus de développement itératifs et incrémentaux. Ainsi, le processus *MoVAL-ADP* s'adapte parfaitement avec les processus itératifs de développement de logiciels tel que le processus unifié (*Unified Process*) comme détaillé dans la section 6.5, ou les processus de développement agiles tel que *Scrum*.

6.4 Les phases de *MoVAL-ADP*

Le processus de définition d'architecture logicielle *MoVAL*, est un processus conforme au processus générique *Unified Process*. Ainsi, ses différentes activités sont regroupées en quatre phases distinctes, illustrées dans la figure 6.6, qui sont : *Inception phase*, *Elaboration phase*, *Construction phase*, *Transition phase*.

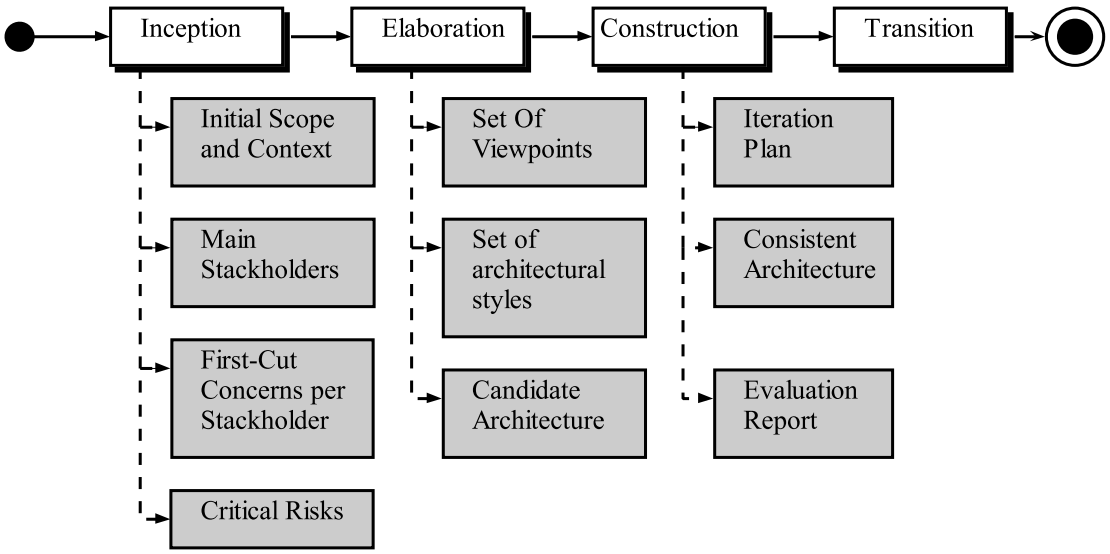


FIGURE 6.6 – Les quatre phases de *MoVAL-ADP*.

6.4.1 La phase *Inception* dans *MoVAL-ADP*

La première phase du processus de définition d'architecture logicielle *MoVAL* est la phase d'*Inception*. C'est une phase d'étude de faisabilité, préparatoire qui aide l'architecte à s'engager dans le processus de définition de l'architecture. Cette phase a pour objectifs de constituer une idée générale à propos du cadre et du contexte du système en considération, et d'accueillir les informations nécessaires pour pouvoir démarrer les autres phases de ce processus, durant lesquelles les différents besoins des différents intervenants vont être identifiés et formalisés pour former l'architecture logicielle de ce système.
Normalement, cette phase peut être couverte par une seule itération, sauf dans des cas exceptionnels de systèmes largement complexes dans lesquelles l'architecte décide de démarrer plusieurs itérations pour recueillir les informations nécessaires et garantir un début de définition d'architecture appropriée répondant aux besoins des différents intervenants concernés par le système informatique.
Cette phase consiste à établir les quatre activités illustrées par le diagramme d'activités de la figure 6.7, et définies dans le tableau 6.1.

6.4.2 La phase *Elaboration* dans *MoVAL-ADP*

La deuxième phase du processus de définition d'une architecture logicielle *MoVAL* est la phase d'élaboration. Cette phase peut être considérée un démarrage pratique du processus de définition de l'architecture, où l'architecte doit consolider toutes les données accueillies pendant la première phase, et essayer d'exposer les solutions architecturales appropriées afin de dégager une architecture logicielle convenable.
Cette phase aussi peut être couverte en passant par plusieurs itérations. Mais souvent dans la plupart des cas des systèmes informatiques, une seule itération sera suffisante pour consolider les données de la première phase et dégager l'architecture candidate.

Dans cette phase, nous définissons les trois activités, illustrées par le diagramme d'activités de la figure 6.8, et définies dans le tableau 6.2.

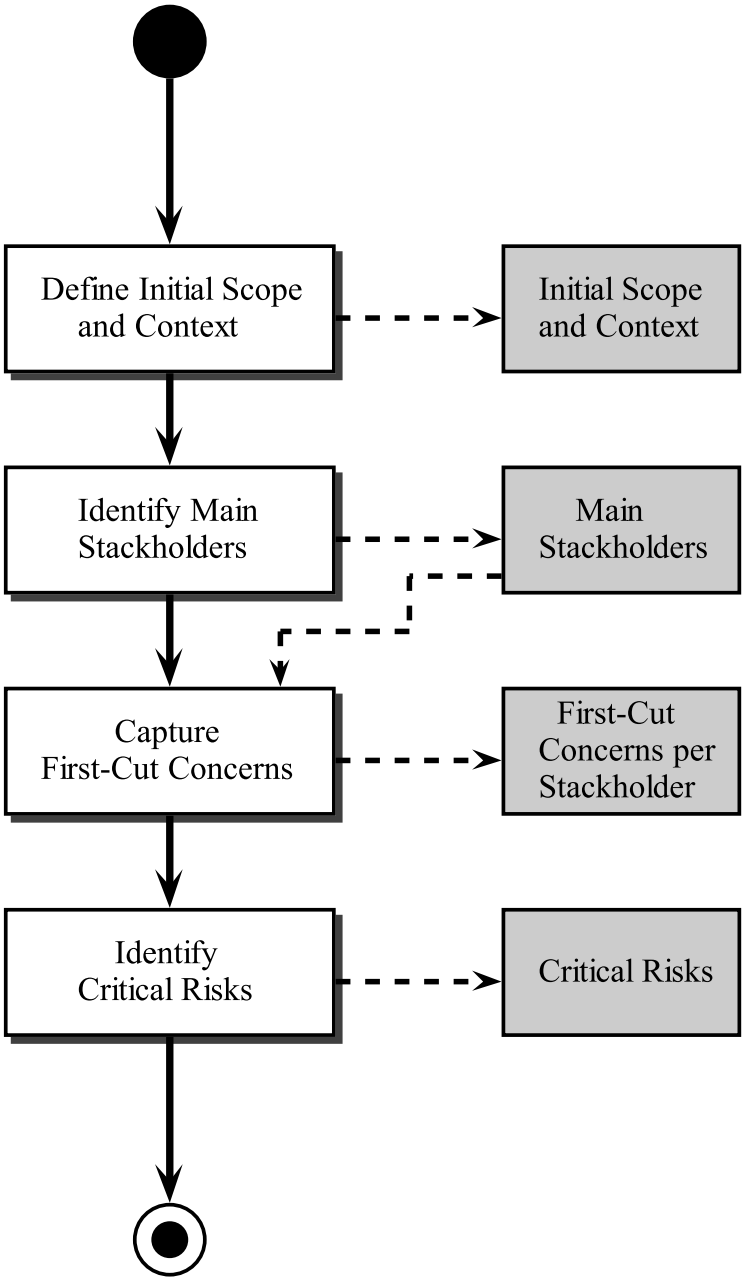


FIGURE 6.7 – *MoVAL-ADP* : Le diagramme d'activités de la phase d'inception (avec les inputs et outputs en gris).

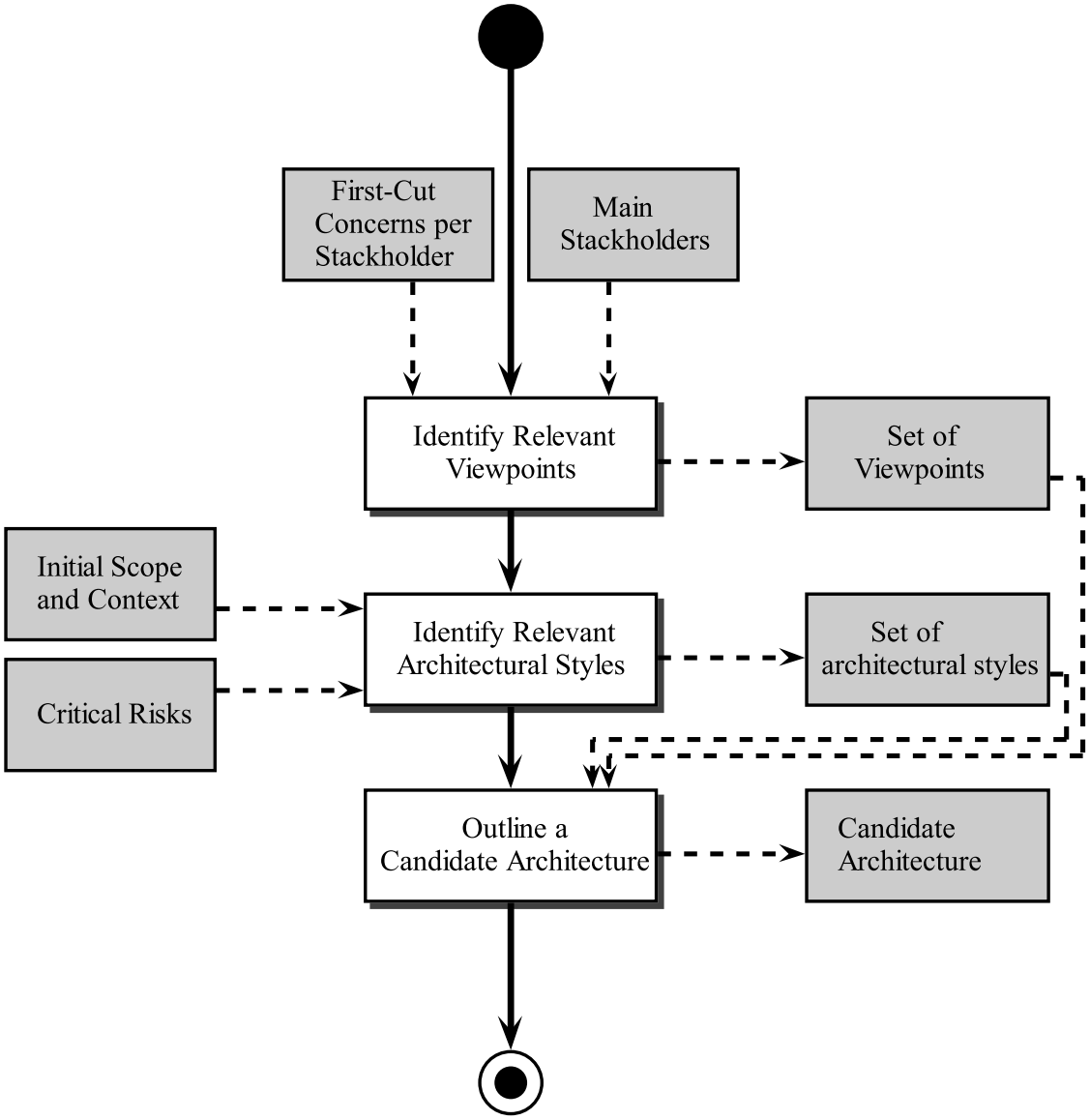


FIGURE 6.8 – *MoVAL-ADP* : Le diagramme d'activités de la phase d'élaboration (avec les inputs et outputs en gris).

TABLE 6.1 – Les activités de la phase *Inception*.

Activité	**Précondition & Objectifs**	**Artefacts produits**
Define initial scope and context	**Objectifs :** Définir les objectifs du système, ses responsabilités, et l'organisation de l'environnement dans lequel il doit agir, en termes de systèmes et dispositifs avec lesquels il doit communiquer.	Cahier des charges initial et réduit.
Identify main stakeholders	**Objectifs :** Identifier les intervenants principaux dans la construction du système.	Liste des intervenants.
Capture first-cut concerns	**Précondition :** Les intervenants et/ou groupe(s) d'intervenants sont connu(s) **Objectifs :** Comprendre les préoccupations globales de chaque intervenant ou groupe d'intervenants.	Cahier des charges plus détaillé.
Identify critical risks	**Objectifs :** Identifier les risques les plus critiques qui peuvent aboutir à des cas de défaillance dans le fonctionnement du système informatique.	Liste des risques.

TABLE 6.2 – Les activités de la phase *Elaboration*.

Activité	**Précondition & Objectifs**	**Artefacts produits**
Identify Relevant Viewpoints	**Précondition :** L'ensemble des intervenants principaux et leurs préoccupations fondamentales sont connus. **Objectifs :** Identifier l'ensemble des points de vue préliminaires de l'architecture logicielle qui seront étendus et divisés après dans les phases prochaines pour couvrir tous les intervenants du système informatique. Alors le principe de cet ensemble est de contenir les points de vue qui peuvent avoir un impact sur l'architecture candidate et sur les lignes directrices de l'architecture logicielle du système en général.	L'ensemble des points de vue préliminaires.
Identify Relevant Architectural Styles	**Précondition :** Le domaine d'application du système informatique, son environnement, ses différents intervenants, et les risques les plus critiques qui menacent son exécution sont connus. **Objectifs :** Identifier les styles architecturaux convenables au système en construction.	Un ensemble de styles architecturaux potentiels.
Outline a Candidate Architecture	**Précondition :** L'ensemble des points de vue préliminaires et les styles architecturaux potentiels sont connus. **Objectifs :** Proposer une architecture logicielle candidate répondant aux *first-cut concerns* capturés durant la première phase du processus de définition de l'architecture, la phase d'*Inception*.	Une architecture candidate.

6.4.3 La phase *Construction* dans *MoVAL-ADP*

En général, la phase de construction dans le processus de définition d'une architecture logicielle est la phase la plus lourde et la plus critique parmi les quatre phases de ce processus, contrairement au processus de développement logiciel dans lequel la phase d'élaboration est considérée la phase la plus critique.
Les entrées de cette phase sont : l'architecture candidate (*Candidate architecture*) qui est le résultat de la phase d'élaboration, ainsi que le rapport d'évaluation (*Evaluation report*) qui est un rapport fourni à la fin d'une itération précédente à la base des commentaires des intervenants concernés.
Cette phase commence par la sélection de l'ensemble des vues qui vont être considérées et construites durant cette itération en se basant sur l'ensemble défini dans l'architecture candidate, puis la sélection d'une vue pertinente de cet ensemble, afin de la construire. Puis, l'architecte logiciel doit intégrer cette vue dans l'architecture logicielle en définissant les correspondances entre les différents niveaux de hiérarchie de cette vue et ceux des autres vues construites auparavant, et en détectant et résolvant les inconsistances qui peuvent avoir lieu au niveau de l'architecture suite à l'addition de la nouvelle vue. Après, l'architecte peut re-sélectionner une autre vue et refaire le même cycle. Finalement, l'architecte effectue deux tests d'évaluation :

- Une *évaluation individuelle* pour détecter les défaillances architecturales, comme les inconsistances entre les différentes vues de l'architecture, ou la manque de correspondances entre les niveaux de hiérarchie des vues de cette architecture ; c'est l'activité "*Assess architecture*".

- Une *évaluation collective* avec les intervenants afin de signaler soit l'ignorance de préoccupations qui sont significatives au niveau architectural, soit un mauvais traitement d'autres préoccupations détectées suite à une discussion entre l'architecte d'une part et les intervenants associés d'une autre part ; c'est l'activité "*Evaluate architecture with stakeholders*".

À noter que, souvent, cette phase sera décomposée en plusieurs itérations de manière à prendre en considération soit un nombre défini de points de vue par itération, ou bien en construisant chaque vue associée à un point de vue sélectionné jusqu'à un niveau de réalisation bien défini, afin qu'elle soit plus détaillée dans les prochaines itérations. De même, les deux types de décomposition en itérations peuvent être appliqués simultanément dans certains cas.

La figure 6.9 représente un diagramme d'activités illustrant les différentes activités composant la phase de construction du processus de définition d'architecture logicielle *MoVAL*. Ces activités sont définies dans le tableau 6.3.

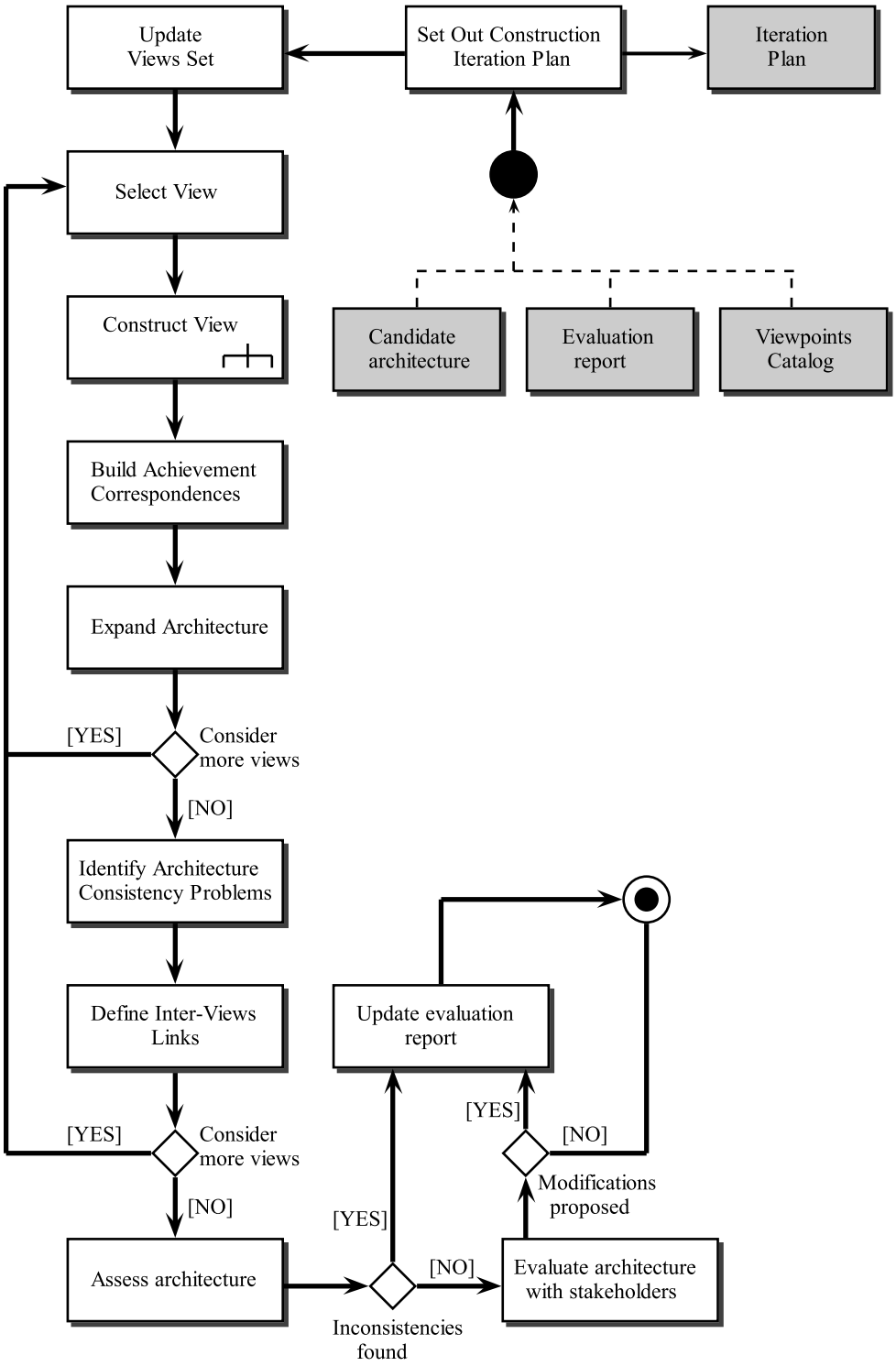


FIGURE 6.9 – *MoVAL-ADP* : Le diagramme d'activités de la phase de construction (avec les inputs et outputs en gris).

TABLE 6.3 – Les activités de la phase *Construction*.

Activité	**Précondition & Objectifs**	**Artefacts produits**
Set Out Construction Iteration Plan	**Précondition :** L'architecture candidate, le rapport d'évaluation, ainsi que le catalogue de point de vue. **Objectifs :** Examiner les points de vue/ les vues qui doivent être ajoutées.	Le plan global de l'itération de construction.
Update Views Set	**Précondition :** La planification globale de l'itération de construction. **Objectifs :** Diviser les vues générales en plusieurs vues plus précises, ajouter de nouvelles vues, ou même supprimer des vues si besoin.	Un ensemble de vues plus précis.
Select View	**Objectifs :** Choisir la vue à construire, ou à détailler en ajoutant d'autres niveaux de réalisation plus avancés.	Une vue précise.
Construct View	**Précondition :** La sélection d'une vue est effectuée. **Objectifs :** Construire la vue sélectionnée. Cette activité sera détaillée dans la section 6.4.3.	L'ensemble de modèle formant la vue considérée, organisés dans des niveaux de hiérarchie inter-liés.
Build Achievement Correspondancies	**Précondition :** L'ensemble des niveaux de hiérarchie de la vue construite. **Objectifs :** Établir les relations de correspondances entre les différents niveaux de réalisation des vues de l'architecture. Les relations de correspondance doivent respecter les règles détaillées dans la section 4.6.2.	L'ensemble des liens de correspondance nécessaires pour intégrer la vue dans l'architecture logicielle.
Expand Architecture	**Précondition :** L'architecte connait l'ensemble de modèles formant la vue considérée, organisés dans des niveaux de hiérarchie inter-liés, ainsi que l'ensemble des liens de correspondance nécessaire pour intégrer cette vue dans l'architecture logicielle. **Objectifs :** Etendre l'architecture logicielle en intégrant la vue nouvellement construite dans cette architecture.	L'architecture logicielle étendue par la nouvelle vue. De plus, les niveaux de réalisation entre les différentes vues sont mis en correspondance.

Activité	Précondition & Objectifs	Artefacts produits
Identify Consistency Problems	**Précondition :** Les définitions détaillées des besoins et des exigences des vues considérées dans l'itération courante et l'architecture obtenue suite à cette itération. **Objectifs :** Identifier les problèmes d'inconsistance qui peuvent avoir lieu au niveau de l'architecture en gros, et cela en détectant les contradictions ou les incohérences entre les besoins et les exigences exprimées et représentées dans plusieurs vues de l'architecture. Normalement cette détection doit être manuelle et se basant sur les expériences individuelles de l'architecte logiciel.	L'ensemble des inconsistances détectées.
Define Inter-Views Links	**Précondition :** La définition de l'ensemble des inconsistances de l'architecture logicielle. **Objectifs :** Définir les liens inter-vues (référence/dépendance) nécessaires pour résoudre les problèmes d'inconsistance.	Une nouvelle version consistante de l'architecture logicielle.
Assess Architecture	**Objectifs :** Assurer que tous les exigences et les besoins des intervenants qui doivent être considérés durant cette itération ont été pris en compte en éliminant les inconsistances qui ont eu lieu entre eux. Cette activité doit être effectuée en général par l'architecte logicielle.	Un ensemble de *feedbacks.*
Evaluate Architecture with Stakeholders	**Objectifs :** Assurer que tous les exigences et les besoins des intervenants qui doivent être considérés durant cette itération ont été pris en compte en éliminant les inconsistances qui ont eu lieu entre eux. Cette activité doit être effectuée en général dans le cadre d'une réunion entre, d'une part, les intervenants concernés par les vues considérées dans cette itération et l'architecte.	Un ensemble de *feedbacks.*
Update Evaluation Report	**Précondition :** Un ensemble de feedbacks détectés. **Objectifs :** Mettre à jour le rapport d'évaluation en ajoutant d'une part, les problèmes détectés dans les activités d'évaluation effectuées, qu'il soit des problèmes d'inconsistances ou bien des problèmes d'insuffisance dans les besoins couverts. D'autre part, en supprimant les problèmes détectés dans des itérations précédentes et fixés dans cette itération.	Un rapport d'évaluation pour l'itération courante.

L’activité "*Construct View*"

L’activité de construction d’une vue de l’architecture logicielle n’est pas une activité réservée uniquement à l’architecte. En effet, c’est une activité collaborative impliquant l’architecte et les équipes de spécification, modélisation, et parfois l’équipe d’implémentation du système informatique.
Cette activité commence par une spécification détaillée des besoins des intervenants impliqués dans la vue en construction, en effectuant des réunions entre ces intervenants, l’architecte et l’analyste. Ensuite, la spécification détaillée sera successivement transformée en un modèle ou un ensemble de modèles ayant des intérêts différents pour couvrir les différents aspects importants dans le cadre de cette vue. Chaque modèle parmi cet ensemble sera raffiné séparément en lui créant d’autres modèles dans d’autres niveaux de réalisation et de description. Ces modèles seront alors organisés physiquement dans des répertoires associés à leurs niveaux de hiérarchie.
Le répertoire est l’endroit où les modèles de l’architecture seront sauvegardés. Il est défini par le triplet : vue, niveau de réalisation et niveau de description.
Finalement, des évaluations vont être effectuées pour détecter des défaillances potentielles au niveau des préoccupations considérées ou au niveau de leur représentation dans le modèle initial, et aussi pour détecter les problèmes d’inconsistances qui peuvent avoir lieu et les résoudre en définissant des liens architecturaux entre les différents niveaux de hiérarchie de cette vue.
Les sous-activités composantes de l’activité "*Construct View*", sont représentées dans le diagramme de la figure 6.10, et sont détaillées dans le tableau 6.4.

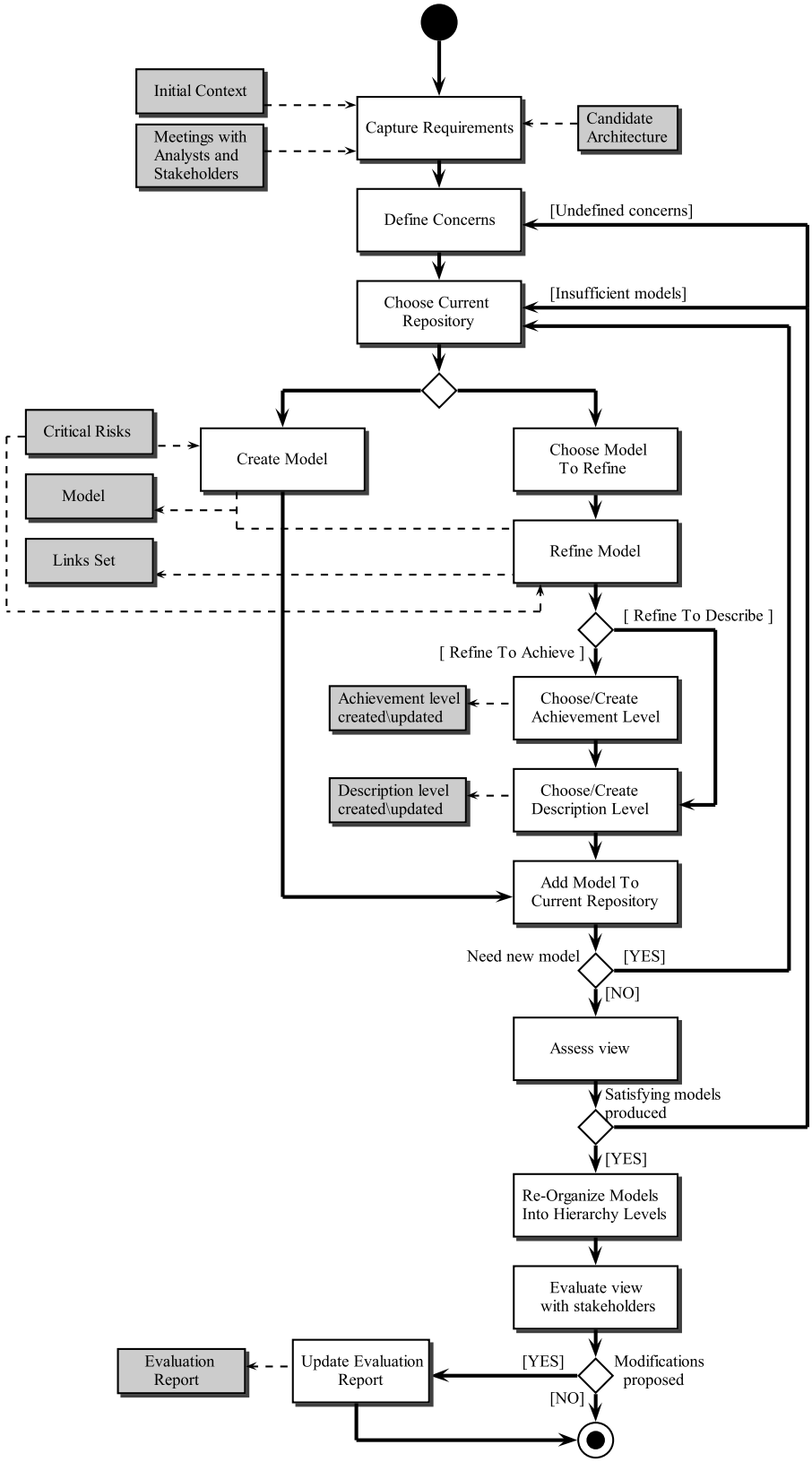


FIGURE 6.10 – *MoVAL-ADP* : Le diagramme d'activités "*Construct View*" (avec les inputs et outputs en gris).

TABLE 6.4 – Les sous-activités de l'activité "*Construct View*".

Activité	**Précondition & Objectifs**	**Artefacts produits**
Capture Requirements	**Précondition :** Éffectuer des réunions entre les intervenants concernés, l'architecte logiciel, et l'analyste des besoins. **Objectifs :** Rassembler les besoins des intervenants associés au point de vue adressé.	Les exigences et les besoins détaillés associés à la vue en construction.
Define Concerns	**Précondition :** Les exigences et les besoins détaillés associés à la vue en construction sont acquis. **Objectifs :** Documenter les besoins acquis dans la première activité.	Une documentation textuelle et formelle des exigences associées à la vue en construction.
Choose Current Repository	**Objectifs :** Choisir le répertoire dans lequel l'architecte cherche à créer un nouveau modèle ou dans lequel se trouve le modèle qu'il cherche à raffiner.	Le répertoire courant.
Create Model	**Précondition :** Les risques les plus critiques en relation avec le scope du modèle sont dégagés durant la phase *Inception*. **Objectifs :** Créer un modèle répondant à toutes/un sous-ensemble des préoccupations détectées lors des deux activités précédentes (*Capture Requirements*, *Define Concerns*).	Un modèle de la vue.
Choose Model To Refine	**Précondition :** Il existe au moins un modèle dans la vue. **Objectifs :** Choisir un modèle afin qu'il sera raffiné dans l'activité suivante.	Le modèle choisi à raffiner.
Refine Model	**Précondition :** Le modèle est choisi dans l'activité précédente. **Objectifs :** Créer un modèle raffiné.	Un modèle raffiné.

Activité	Précondition & Objectifs	Artefacts produits
Choose/Create Achievement Level	**Objectifs :** Choisir/créer un niveau de réalisation.	Le niveau de réalisation.
Choose/Create Description Level	**Précondition :** Le niveau de réalisation est choisi. **Objectifs :** Choisir/créer un niveau de description.	Le niveau de description.
Add Model To Current Repository	**Précondition :** Le modèle, le niveau de réalisation, et le niveau de description sont connus. **Objectifs :** Sauvegarder le modèle dans le répertoire identifié par la vue courante et les niveaux en données.	L'ensemble des modèles de la vue est étendu par le nouveau modèle.
Assess View	**Objectifs :** Assurer que tous les exigences et les besoins des intervenants qui doivent être considérés durant cette itération ont été pris en compte et correctement représentés dans les modèles créés. Cette activité doit être effectuée en général par l'architecte.	Un ensemble de *feedbacks*.
Re-Organize Models into Hierarchy Levels	**Objectifs :** Réviser l'organisation des modèles obtenus et définis pour la vue courante, et effectuer les modifications nécessaires sur cette organisations qui ont été détectées durant l'activité (*Assess View*). **Remarque :** À noter que la position d'un modèle ne peut jamais être modifiée si ce modèle possède déjà des liens avec d'autres modèles.	Une organisation plus précise des modèles dans les niveaux de hiérarchie de la vue.
Evaluate View with Stakeholders	**Objectifs :** Assurer que tous les exigences et les besoins des intervenants qui doivent être considérés durant cette itération ont été pris en compte et correctement représentés dans les modèles créés. Cette activité doit être effectuée en général dans le cadre d'une réunion entre, d'une part, les intervenants concernés par les vues considérées dans cette itération et l'architecte.	Un ensemble de *feedbacks*.
Update Evaluation Report	**Précondition :** Un ensemble de feedbacks détectés. **Objectifs :** Mettre à jour le rapport d'évaluation en ajoutant d'une part, les problèmes détectés dans les activités d'évaluation effectuées, qu'il soit des problèmes d'inconsistances ou bien des problèmes d'insuffisance dans les besoins couvertes. D'autre part, en supprimant les problèmes détectés dans des itérations précédentes et fixés dans cette itération.	Un rapport d'évaluation pour l'itération courante.

6.4.4 La phase *Transition* dans *MoVAL-ADP*

La dernière phase du processus de définition de l'architecture logicielle *MoVAL* est la phase de transition, parfois aussi nommée la phase "*Fine tuning*", qui peut être considérée la phase la plus longue, mais en même temps la plus "légère" en terme d'effort et de temps de travail, parmi les quatre phases du processus *MoVAL-ADP*.
En effet, cette phase s'effectue conjointement avec les phases de construction et de transition du processus de développement logiciel. Elle est ponctuée par des "légers raffinements" de l'architecture à la fin de chaque itération où sont collectés les *feedbacks* de l'équipe de développement et des utilisateurs du système informatique. Dans le cas où les trois phases précédentes du processus *MoVAL-ADP* ont été bien élaborées, nous espérons que les raffinements effectués dans cette phase sont minimes et ne sont pas des opérations de mise en question des décisions majeurs prises auparavant.
Le but ultime de cette phase, est de maintenir l'architecture logicielle à jour tout le long du processus de développement. Ainsi, la description de l'architecture resterait le document de base, et le support robuste auquel on a recours en cas d'une évolution ou de maintenance.
De plus, cette phase permet à l'architecte logiciel de contrôler les phases de construction et de transition du processus de développement logiciel. En cas de lapsus au niveau de l'architecture en cours de développement, l'architecte peut ainsi intervenir pour prendre les décisions architecturales nécessaires pour assurer la qualité architecturale du système.

6.5 La position du *MoVAL-ADP* par rapport au processus de développement

Comme dit précédemment, le processus de définition d'une architecture logicielle (*MoVAL-ADP*) s'adapte parfaitement avec le processus de développement logiciel (*SDP*). Ces deux processus sont concurrents interagissant ensemble et s'échangant des flux d'informations durant leur existence. Nous avons étudié l'utilisation d'un *SDP* incrémental et itératif tel le *RUP* conjointement avec *MoVAL-ADP*.
La figure 6.11 illustre cette concurrence entre ces deux processus et présente l'effort dépensé sur la définition de l'architecture durant les différentes phases des deux processus.

Comme on peut remarquer, les deux phases d'*Inception* et d'*Elaboration* du processus de définition de l'architecture logicielle se situent ensemble exactement en coincidence avec la phase d'*Inception* du processus de développement logiciel, mais ces deux phases se terminent directement avant la phase *Inception* du *SDP* puisque cette dernière doit fournir l'architecture candidate produite à la fin de la phase *Elaboration* du *ADP*. Pourtant, la phase de *Construction* du processus de définition de l'architecture logicielle se situe en coincidence avec la phase d'*Elaboration* du processus de développement logiciel. Finalement, la phase de *Transition* du processus de définition de l'architecture logicielle se situe en coincidence avec les deux phases de *Construction* et de *Transition* du processus de développement logiciel.
Egalement, on peut remarquer dans cette figure, que l'effort dépensé sur la définition de l'architecture est maximal durant les trois premières phases du processus de définition de l'architecture, à savoir, durant les phases d'*Inception* et d'*Elaboration* du processus de développement logiciel. Cependant, cet effort s'abaisse significativement durant toutes les itérations des phases de *Construction* et de *Transition* du processus de développement, et remonte un petit peu pour une durée très limitée à la fin de chaque itération. Normalement, cela signifie qu'à la fin de chaque itération de ces deux phases du processus de développement, l'architecte logiciel dépense un petit effort à mettre à jour son architecture logicielle en se basant sur les *feedbacks* reçus de l'équipe de développement, et reste en attente pendant que cette équipe soit en plein travail pendant les itérations.

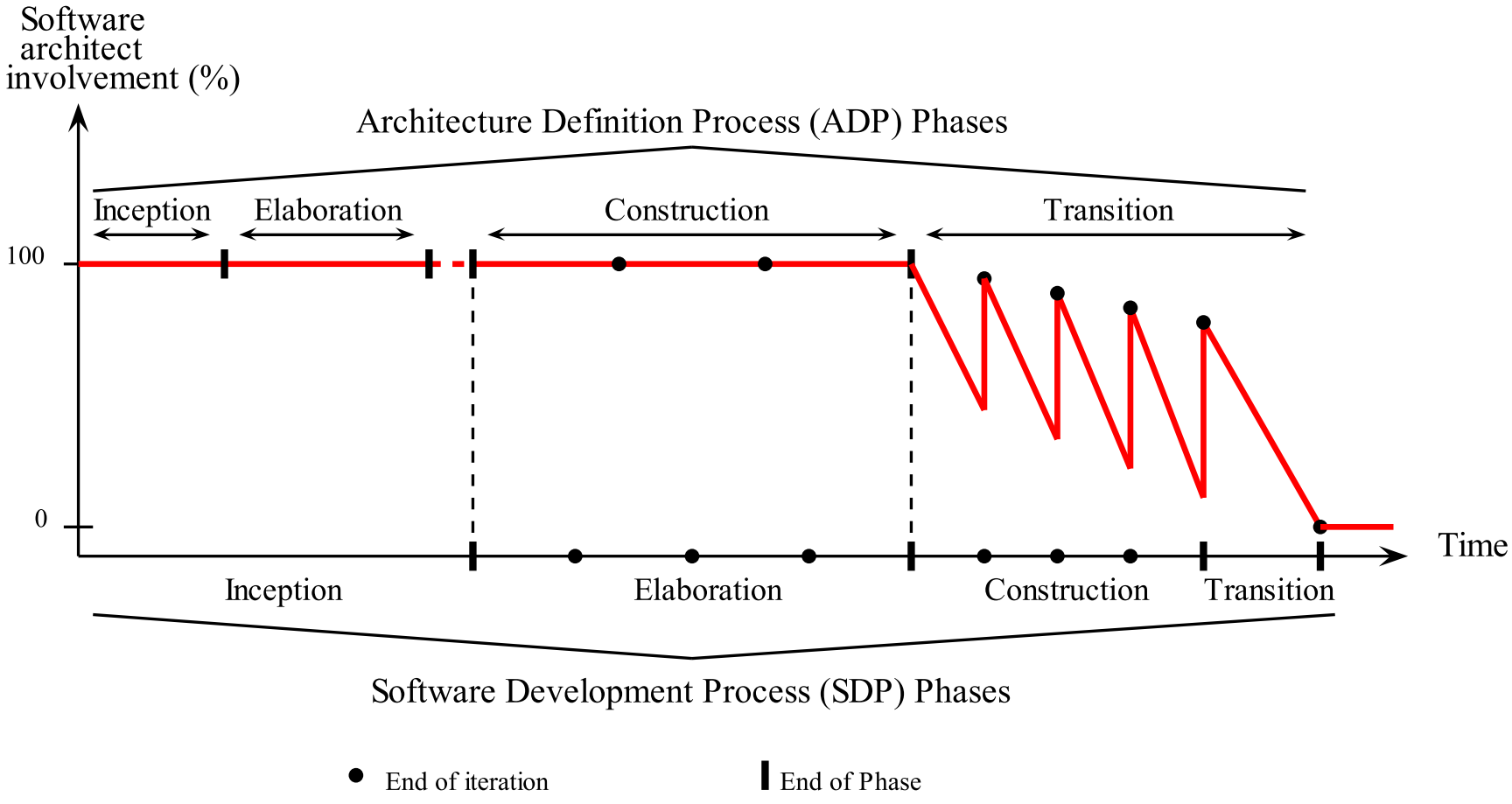


FIGURE 6.11 – Les niveaux de participation de l'architecte logiciel au cours des différentes phases.

TABLE 6.5 – Les résultats des activités de la phase Inception du système *SWVE*.

Activité	Résultats
1- Define initial scope and context	Le système en considération fourni une application Web pour les clients afin de les permettre de faire des achats en ligne parmi un catalogue de produits exposés en ligne.
2- Identify main stakeholders	Client, Agent de services clientèles, Administrateur du catalogue, architecte logiciel, analyste logiciel, Programmeur Web, Programmeur d'applications bureautiques, Administrateur des bases de données, Ingénieure du réseau, etc. ...
3- Capture first-cut concerns	• Le client doit être capable de faire des achats sur Web • L'agent des services clientèles doit être capable de gérer les clients à travers une application bureautique spécifique • L'administrateur du catalogue doit être capable de gérer l'ensemble des produits à travers une interface Web locale • Les demandes doivent être envoyées à un service extérieur pour qu'elles soient manipulées • ...
4- Identify critical risks	• Le vol des données des clients • Demandes passées avec succès localement, mais ratées par le service extérieur • ...

6.6 Application de *MoVAL-ADP* sur le cas d'étude

Afin de démontrer l'applicabilité de *MoVAL-ADP* sur des cas d'étude pratiques, nous allons l'appliqué dans cette section sur le système *SWVE* qu'on a introduit auparavant dans la section 4.3 du chapitre 4. Le tableau 6.5 représente le cadre et le contexte du système *SWVE*, ces intervenants principaux, leurs préoccupations fondamentales, et les risques les plus critiques de ce système. Donc, ce tableau représente les activités de la phase *Inception* de *MoVAL-ADP*. Le tableau 6.6 représente les résultats des différentes activités de la phase d'élaboration du processus *MoVAL-ADP* associée au système *SWVE*.

Maintenant, revenons à la vue fonctionnelle du système *SWVE* déjà introduite dans la section 4.5.2 du chapitre 4, afin d'illustrer les activités principales de la phase *Construction* du processus *MoVAL-ADP*. Ainsi, le diagramme initial informel de cette vue représenté dans la figure 4.3 a été raffiné et concrétisé pour obtenir le diagramme représenté dans la figure 4.4.
Alors, ce modèle représente les même sémantiques représentées dans le diagramme informel mais en utilisant une notation un peu plus technique qui est la notation de composants logiciels.
De même, la figure 4.5 représente un autre modèle qui étend celui de la figure 4.4, en conservant la même notation et représente un niveau de description un peu plus élevé.

TABLE 6.6 – Elaboration phase activities results

Activité	Résultats
1- Identify Relevant Viewpoints	• Fonctionnel • Physique • Information • Sécurité • ...
2- Identify Relevant Architectural Styles	Services Web
3- Outline a Candidate Architecture	Normalement on ne va pas considérer l'architecture logicielle tout entière du système *SWVE*, on va juste considérer les points de vue fonctionnel et physique. Chaque point de vue est représenté par un seul modèle : • le modèle informel de la figure 4.3 pour le point de vue fonctionnel ; • et le modèle informel de la figure 4.2 pour le point de vue physique. Comme nous remarquons, ces modèle utilise des notations et des formalismes informels afin d'être compréhensible par les intervenants adressés durant cette phase du processus de définition de l'architecture logicielle. Puisque, souvent, ces intervenants ne sont pas des techniciens, et alors ne veulent pas, ou même ne sont pas capable, de comprendre des notations et des formalismes ayant des niveaux techniques plus élevés.

6.7 Conclusion

Un processus de définition d'architecture logicielle est un regroupement d'activités qui mènent à la construction et la documentation d'une architecture logicielle. Dans ce chapitre, nous avons présenté le processus de définition d'architecture proposé par *Rozanski* et *Woods*, le processus de développement unifié, et quelques processus de la génération agile.
Ensuite, se basant sur ces processus, nous avons proposé un processus de définition d'architecture logicielle propre à *MoVAL*, dénoté *MoVAL-ADP*, qui guide l'architecte logiciel à définir tous les éléments nécessaires de l'architecture, et lui propose un plan itératif et incrémental pour cette définition. En effet, *MoVAL-ADP* est conforme au processus unifié, et donc divisé en quatre phases différentes qui sont : la phase *Inception*, *Elaboration*, *Construction*, et *Transition*. Finalement, nous avons expliqué la position de notre processus par rapport au processus de développement logiciel unifié. Enfin, nous avons appliqué le processus proposé sur le cas détude *SWVE*.

Expérimentations

7

Validation de l'approche

7.1 Introduction

Dans les chapitres précédents une approche formelle d'architecture logicielle, dénotée *MoVAL*, a été introduite et détaillée, puis mise en place à travers une méthodologie et un processus de définition d'architecture pour guider son application. En plus, un cas d'étude d'un système Web de ventes électroniques a été considéré tout le long de ces chapitres afin de valider l'utilisabilité et l'efficacité de cette approche.
Dans ce chapitre, nous allons introduire des méthodes d'implémentation et de prototypage de *MoVAL*. En effet, *MoVAL* est une approche théorique en premier place, tenant compte des aspects pratiques et industriels, ce qui lui permet de se promouvoir, par un prototype industriel approprié, de son état théorique en une plateforme pratique, complète, fiable et pertinente pour les membres et les équipes de développement logiciel industriels.
Normalement, *MoVAL* est une approche de définition d'architecture logicielle étendue pour soutenir le processus de développement tout entier, comprenant ses différentes phases depuis la phase de spécification des besoins, la conception, jusqu'à la modélisation. D'où la nécessité d'avoir un outil permettant la définition de l'architecture globale des systèmes informatiques et permettant de même la définition des modèles spécifiques du système, comme par exemple les modèles *UML*. Pour cette raison, nous avons conçu une application bureautique, dénotée *MoVAL-Tool*, pour la gestion des architectures logicielles suivant l'approche proposée dans *MoVAL*.
Dans la suite, nous allons présenter dans la section 7.2 une justification de notre choix d'implémentation du application autonome pour l'outil *MoVAL-Tool*. Puis, nous présentons le cadre de la réalisation de cet outil dans la section 7.3. Ensuite, dans la section 7.4, nous présentons les fonctionnalités offertes par notre outil et nous illustrons ces fonctionnalités par des captures d'écran. Finalement, la section 7.5 conclut le chapitre.

7.2 Justification du choix de type d'implémentation

L'objectif principal du prototype d'implémentation associé à l'approche *MoVAL*, est d'avoir un outil permettant la construction d'une architecture logicielle bien documentée suivant la proposition menée dans les chapitres précédents de cette thèse. Afin d'aboutir à cet objectif, nous avons à choisir parmi trois possibilités de types d'implémentation qui sont : les *Profiles UML*, les langages spécifiques aux domaines, et une application autonome.

7.2.1 Les profiles UML

La notion de profiles proposée dans le cadre du "*Meta-Object Facility*" (*MOF*) permet l'extension d'un méta-modèle de base comme le méta-modèle *UML*, et sa personnalisation pour des plateformes (*J2EE/EJB*, *.NET/Com+*) ou des domaines (Finance, Télécommunication, etc. ...) particuliers.
Alors, le mécanisme des profiles permet l'héritage des méta-éléments d'un méta-modèle pour construire de nouvelles méta-classes et méta-associations en ajoutant à ceux du méta-modèle de base de nouvelles contraintes nécessaires pour une plateforme ou un domaine particulier, sans pouvoir effectuer des changements sur le méta-modèle de base, comme par exemple l'addition de certaines nouvelles méta-classes ou méta-associations.
Par la suite, un modèle déclare l'application d'un profile donné pour une seule fois sans avoir besoin d'associer chaque élément de ce modèle au stéréotype du profile. Donc, les profiles peuvent être utilisés normalement conjointement avec l'infrastructure du méta-modèle de base, particulièrement l'infrastructure *UML*.
En effet, ce mécanisme peut être utilisé pour plusieurs raisons :

- donner une terminologie adaptée à une plateforme ou un domaine particulier ;
- donner de nouvelles notations ou modifier des notations existantes ;
- ajouter des sémantiques supplémentaires au méta-modèle de base dans le cadre de ce profile ;
- ajouter de nouvelles contraintes qui restreignent l'utilisation des éléments du méta-modèle ;
- ajouter de nouvelles informations qui peuvent être acquis ultérieurement dans le modèle.

7.2.2 Les langages spécifiques aux domaines

La notion théorique des langages spécifiques aux domaines (*Domain-Specific Languages*, *DSL*) n'a pas été largement diffusée et connues qu'après sa mise en place par *Microsoft* dans sa plateforme *.NET* au début des années 2000.
Un *DSL* est un langage conçu pour être utilisé pour des tâches spécifiques et dans des domaines particuliers, afin de minimiser les efforts dépensés dans l'accomplissement de ces tâches et améliorer leur qualité et leur communication avec l'audience intéressée. Ainsi, un *DSL* peut être assimilé aux diagrammes *UML* qui représentent un langage spécifique au domaine de développement logiciel. Ces langages ciblent alors l'orientation des langages de modélisation vers des domaines, des technologies, et des plateformes beaucoup plus spécifiques.
Normalement, l'avantage ultime de ces langages reste toujours leur simplicité et leur indépendance de tout méta-modèle ou langage.
Les langages spécifiques aux domaines ont été implémentés dans plusieurs environnements de développement comme les environnements *eclipse* et *.NET* , ou dans des outils spécifiques à l'instar de l'outil *MetaCase*. Comme déjà mentionné au-dessus, les langages spécifiques aux domaines ont connu un succès important suite à leur implémentation dans la plateforme *.NET*. Cette plateforme offre les outils nécessaires pour pouvoir créer à partir d'un *DSL* des modèles graphiques personnalisés. En plus, elle permet de créer des patrons textuels pour générer du code à partir de ces modèles.
De même, l'environnement *eclipse* propose dans son projet *EMF* (*Eclipse Modeling Framework*) un outil pour créer un langage spécifique au domaine. Cela en créant tout d'abord un modèle *ecore* pour la définition des concepts du langage, puis en ajoutant un éditeur graphique ou textuel permettant aux utilisateurs de modéliser graphiquement ou textuellement des problèmes de ce domaine. En plus, *eclipse* offre des mécanismes pour permettre la génération du code à partir des modèles créés par ce langage.
Egalement, parmi les outils *DSL* indépendants les plus importants on peut citer *MetaCase*. Cet outil permet aux développeurs de créer leurs langages spécifiques aux domaines en définissant tout d'abord les concepts et les règles du langage, puis les règles de leur transformation en code.

7.2.3 Les applications autonomes

Une application autonome, mieux connue en anglais sous le nom *standalone application*, est une application développée pour atteindre des objectifs et des fonctionnalités spécifiques, sans qu'elle soit dépendante pour son exécution d'un autre système. En général, ce type d'applications offre une grande marge de personnalisation puisque ces applications n'ont pas à respecter des contraintes apposées par un environnement extérieur spécifique.

7.2.4 Une comparaison entre les choix possibles

Pour pouvoir choisir la meilleure approche d'implémentation pour valider l'approche *MoVAL*, nous allons introduire dans cette section une petite comparaison entre les trois choix de méthodes d'implémentation qui sont : les *Profiles UML*, les langages spécifiques aux domaines et une application autonome.
Cette comparaison porte sur six caractéristiques significatives sous l'optique de nos objectifs de prototypage :

- La simplicité de développement :
  La simplicité de développement d'un langage ou d'un méta-modèle est une caractéristique significative pour les concepteurs des outils de développement, ou *toolsmith* en anglais. En général, le développement des *Profiles UML* est le plus simples par rapport au développement des *DSL* et des applications autonomes, cela revient au fait qu'un profile étend un méta-modèle existant et réutilise ses définitions et notations, tandis que le développement des *DSL* et des applications autonomes commence à partir de rien, ou "*from scratch*" ;

- La simplicité d'utilisation :
  La simplicité d'utilisation d'une application, d'un langage ou d'un méta-modèle revient principalement au fait d'avoir juste les éléments et les contraintes nécessaires au domaine spécifique comme éléments et contraintes du langage, et le fait de pouvoir effectuer les opérations nécessaires d'une manière flexible. De cette perspective, les applications autonomes offrent une utilisation beaucoup plus simple et aisée pour les modélisateurs des systèmes vis-à-vis des autres méthodes ;

- La précision :
  La précision du langage exprime sa représentation des exigences du domaine particulier tel qu'elles sont sans ajouter d'autres exigences non-reliées à ce domaine et sans manquer d'autres exigences qui ne peuvent pas être représentées. Ainsi, les applications autonomes sont beaucoup plus précises que les *Profiles* et les *DSL*.

- L'intégrité dans l'environnement de développement :
  Les *Profiles UML* et les *DSL* sont intégrables dans l'environnement de développement d'*eclipse*, qui est l'environnement le plus important à base de *Java*, à travers le projet *EMF*. Tandis que, juste les *DSL* sont intégrables dans l'environnement de développement *.NET*. Pourtant ce n'est pas facile de pouvoir développer une application autonome intégrable dans l'un des environnements de développement mentionnés au-dessus.

- La familiarité :
  La familiarité des équipes de développement représente un point fort pour les *Profiles*, puisque qu'ils sont toujours à la base d'un méta-modèle connus auparavant par les architectes logiciels. Cependant, les *DSL* et les applications autonomes ne sont pas déjà connus par les architectes.

- L'habilité à couvrir entièrement l'approche *MoVAL* :
  Seules les applications autonomes possèdent l'habilité à couvrir entièrement l'approche *MoVAL*, comme la définition de la hiérarchie de l'architecture et ses éléments, la définition des configurations

TABLE 7.1 – Tableau récapitulatif des préférences entre les trois approches.

Caractéristique	***Profiles UML***	***DSL***	Application autonome
Simplicité de développement	✓		
Simplicité d'utilisation			✓
Précision			✓
L'intégrité dans l'env. de dév.		✓	
Familiarité	✓		
Habilité à couvrir entièrement l'approche *MoVAL*			✓

architecturales, la définition des catalogues de points de vue et le passage à travers les différentes activités du processus de définition proposé.

Le tableau 7.1 récapitule les préférences entre les trois approches, *Profiles UML*, *DSL* et applications autonomes, pour chaque caractéristique.

Suite à la comparaison du tableau précédent (Tableau 7.1), le choix entre les *Profiles UML*, les langages spécifiques aux domaines (*DSL*) et les applications autonomes été relativement clair avec trois coches pour les applications autonomes, deux coches pour les *Profiles UML* et une seule pour les *DSL*. Ainsi, nous avons décidé d'implémenter notre prototype d'outil *MoVAL-Tool* dans une application autonome.

7.3 Le cadre de la réalisation de *MoVAL-Tool*

L'outil *MoVAL-Tool* est implémenté sous la plateforme de développement de *Microsoft*, la plateforme *.NET 4.0*, en utilisant le langage de programmation *C#*. En plus, l'interface graphique de cette application est construite en utilisant une librairie de contrôle graphique dénotée *DevExpress v2010 vol 1.4*. Les informations utilisées dans l'application sont sauvegardées essentiellement sous la forme de plusieurs fichiers *xml*.

7.4 Fonctionnalités de l'outil *MoVAL-Tool*

Le but de *MoVAL-Tool* est normalement de donner à l'architecte logiciel et à l'équipe de développement un outil pour appliquer les concepts de base de *MoVAL* et de créer une architecture logicielle hiérarchique multipoints de vue selon le processus de définition d'architecture proposé dans cette approche.
Ainsi, *MoVAL-Tool* implémente les fonctionnalités représentées dans le diagramme des cas d'utilisation de la figure 7.1, comme la création d'un catalogue de points de vue, la création d'une architecture logicielle, la visualisation de l'architecture, la construction de l'architecture, etc.

7.4.1 Créer/Modifier un catalogue

Créer un catalogue de points de vue en lui définissant pour chaque point de vue son nom, idée générale, préoccupations, intervenants et formalismes. Les points de vue de ce catalogue seront ensuite sauvegarder en format *XML*, et pourrons être réutilisés sur d'autres machines. La figure 7.2 illustre l'interface graphique offerte par *MoVAL-Tool* pour ce but.
Cette interface offre de plus la possibilité d'importer des catalogues existants pour les modifier et les re-sauvgarder dans un nouveau fichier.

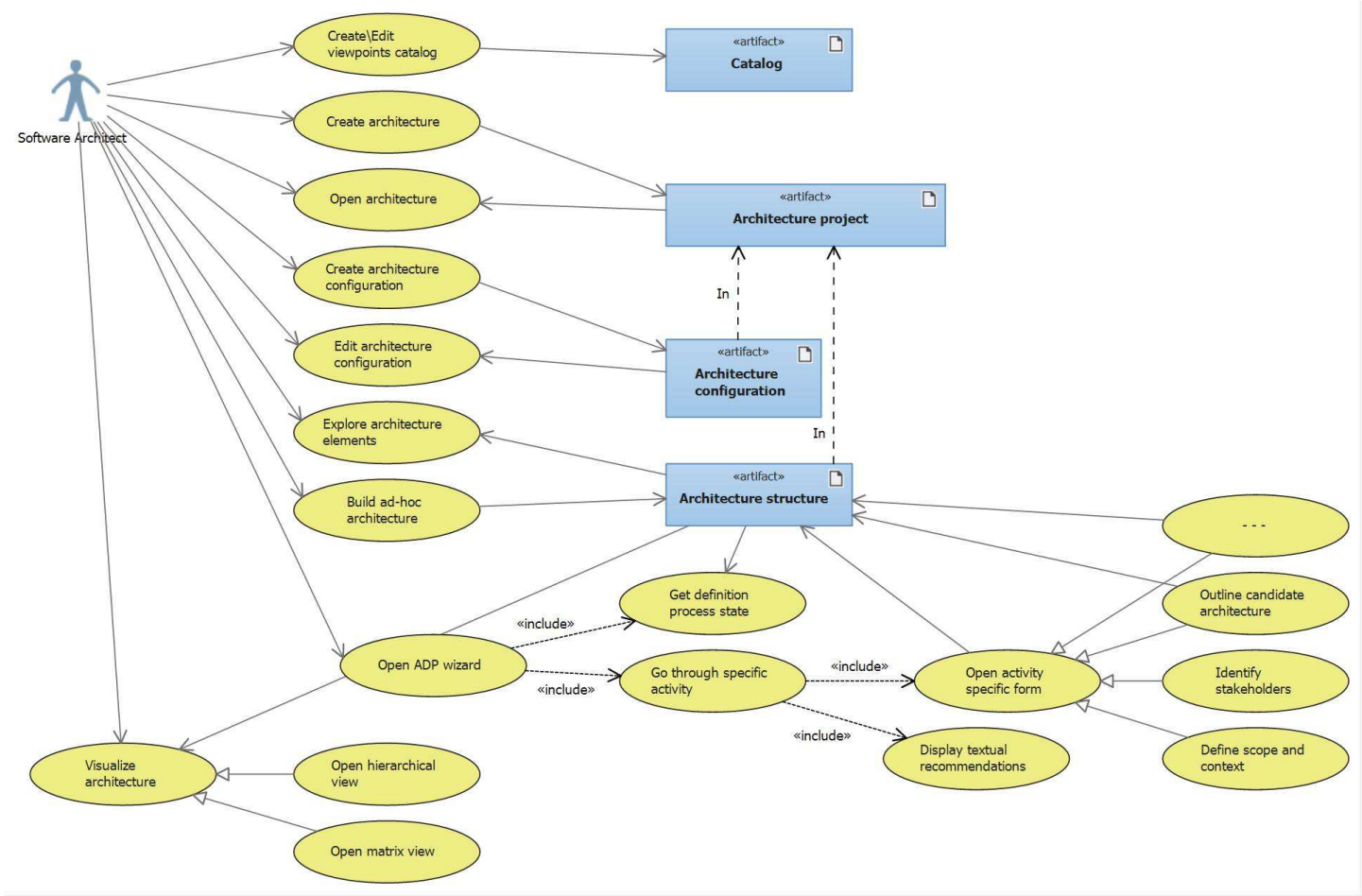


FIGURE 7.1 – Le diagramme UML des cas d'utilisation de *MoVAL-Tool*.

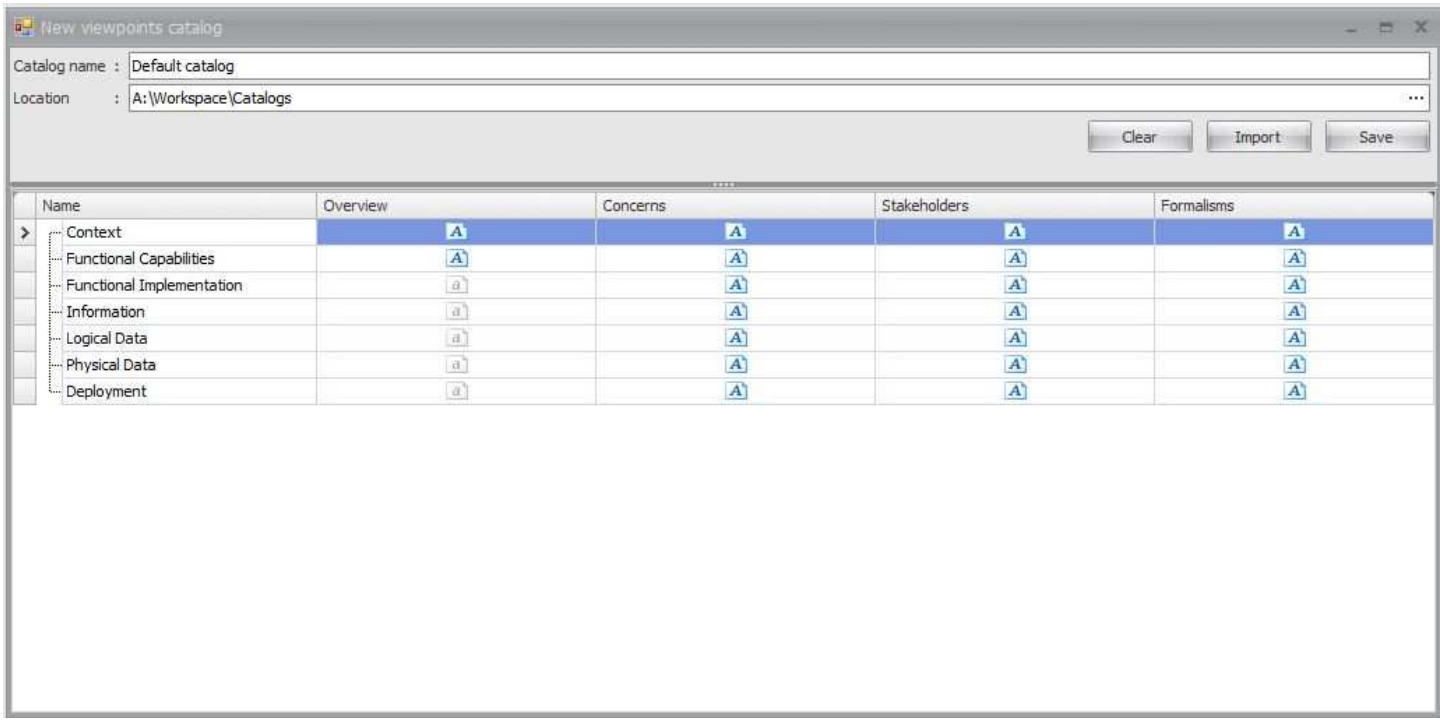


FIGURE 7.2 – La création d'un catalogue de points de vue dans *MoVAL-Tool*.

7.4.2 Créer une architecture

Créer une nouvelle architecture, en lui donnant tout d'abord les informations spécifiques, comme le nom de l'architecture, le système auquel elle est associée, sa version, le nom de l'architecte responsable, etc. Ensuite, l'application doit créer un nouveau répertoire dans lequel elle doit sauvegarder les fichiers spécifiques à cette architecture, et un fichier ayant l'extension ".moval" contenant les informations spécifiques à cette architecture et les chemins (*i.e. paths*) des fichiers associés à cette architecture.
Normalement, on peut toujours ouvrir un projet d'architecture logicielle existant pour continuer le travail

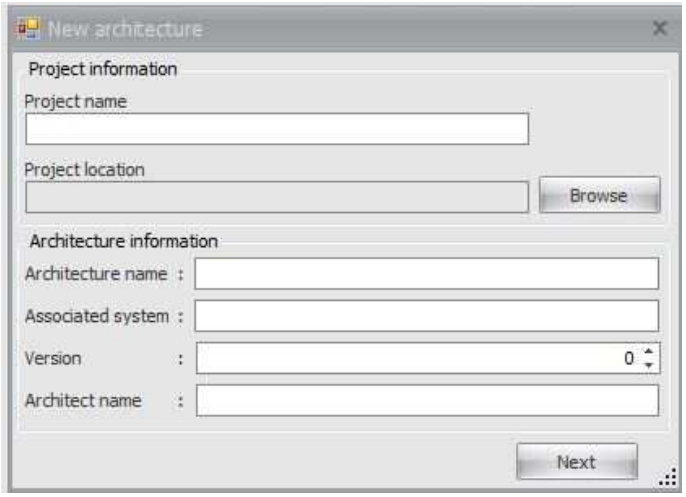


FIGURE 7.3 – La création d'un nouveau projet d'architecture logicielle dans *MoVAL-Tool*.

déjà commencé, et cela en appuyant sur le bouton "*Open architecture*" du menu "*File*" comme illustré dans la figure 7.4.

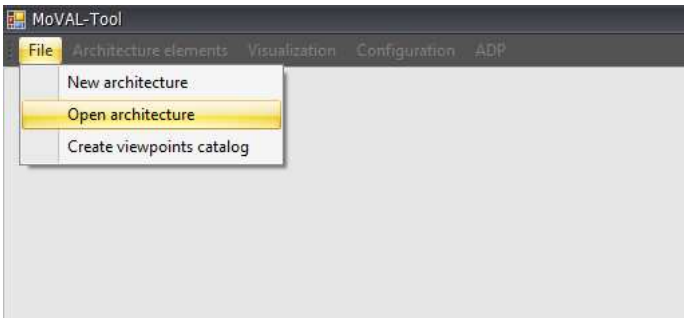


FIGURE 7.4 – L'ouverture d'un projet d'architecture logicielle existant dans *MoVAL-Tool*.

7.4.3 Construire une architecture ad-hoc

Construire graphiquement une architecture ad-hoc, en créant des vues, des niveaux de réalisation, des niveaux de description, des modèles, les liens nécessaires entre les entités et les correspondances entre les différents niveaux de réalisation. Cela à travers un éditeur graphique, illustré dans la figure 7.5.
Dans cet éditeur on trouve une fenêtre à gauche, dénotée "*Tools*" contenant les différents types d'éléments d'une architecture logicielle qui peuvent être ajoutés explicitement par l'architecte, qui sont : (1) une vue, (2) un niveau de réalisation, (3) un niveau de description, (4) un modèle, (5) un lien is_R et finalement un lien de correspondance $is_=$. Un élément de l'architecture est créé en appuyant premièrement sur le type voulu dans la fenêtre des outils "*Tools*" puis en appuyant sur les éléments associés à l'élément créé. Egalement, on trouve une autre fenêtre à droite, dénotée "*Properties*" représentant les propriétés de l'élément architectural sélectionné.

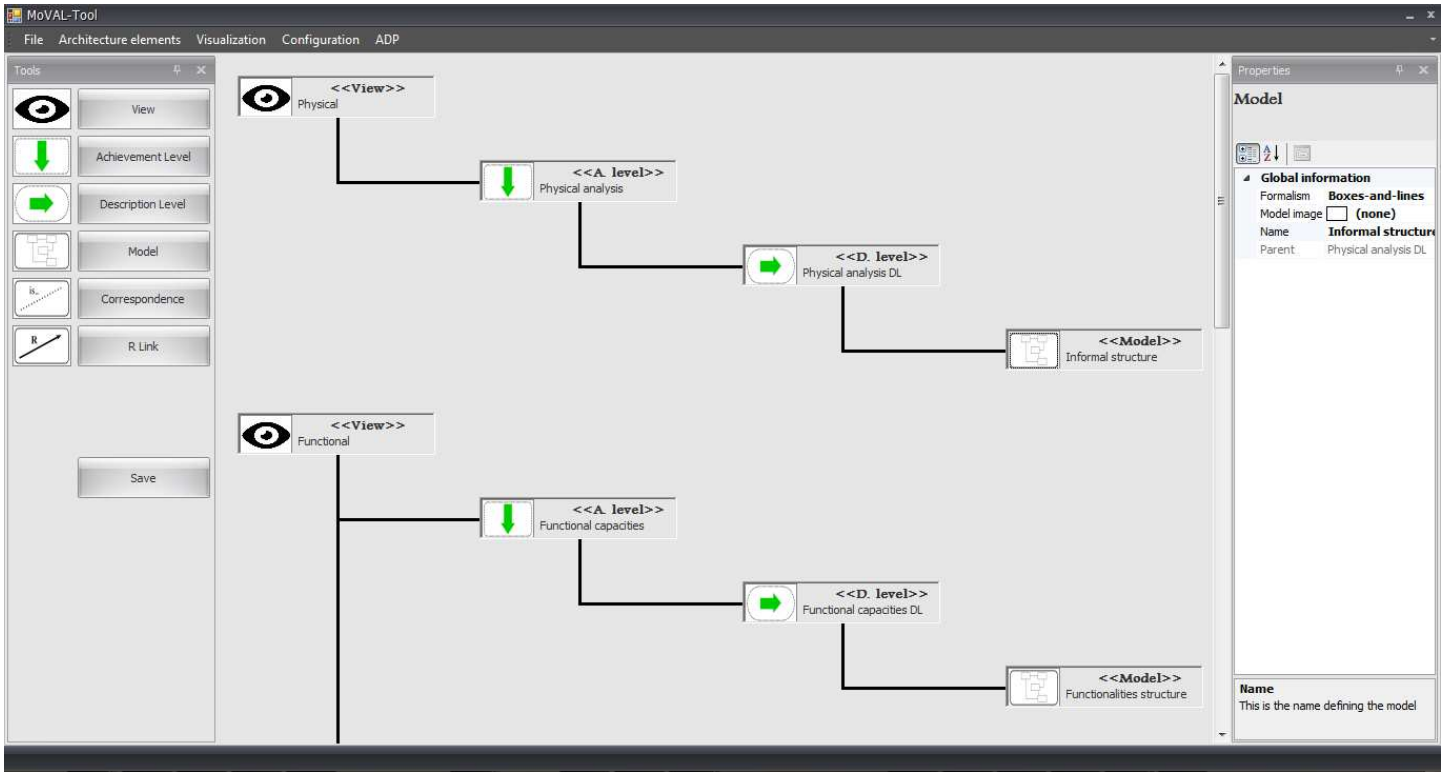


FIGURE 7.5 – L'interface graphique de l'éditeur d'une architecture dans *MoVAL-Tool*.

En fait, cet éditeur peut être utilisé en mode "Pleine fonctionnalité" pour construire une architecture logicielle ad-hoc sans passer par les différentes étapes du processus de définition d'une architecture logicielle *MoVAL-ADP*, et peut être aussi accédé depuis plusieurs étapes de l'*ADP Wizard* (voir section 7.4.7) avec une restriction sur les fonctionnalités offertes selon l'étape invoquant.

7.4.4 Explorer l'arborescence de l'architecture

Explorer les différents constituants de l'architecture sous une forme arborescente, comme les vues, niveaux de réalisation, niveaux de description, modèles, rapports d'évaluation, etc.
La figure 7.6 illustre l'interface créée pour ce but. Dans cette interface l'architecte logiciel peut sélectionner n'importe quel constituant de l'architecture pour visualiser son contenu s'il est un document textuel, pour visualiser une description à propos de son contenu s'il est un élément architectural (vue, niveau de réalisation ou niveau de description), et dans le cas d'un modèle pour visualiser l'image décrivant le contenu de ce modèle.

7.4.5 Créer/Modifier une configuration architecturale

Créer une configuration d'architecture en donnant les informations nécessaires à propos de cette configuration, comme son nom, description, les intervenants adressés, et en sélectionnant les éléments qui doivent être représentés dans cette configuration comme illustré dans la figure 7.7.
La grille illustrée dans la figure 7.8 liste l'ensemble des configurations architecturales créées pour une architecture logicielle. Chacune parmi ces configurations peut être modifiée en double cliquant la ligne associée à cette configuration dans la grille.

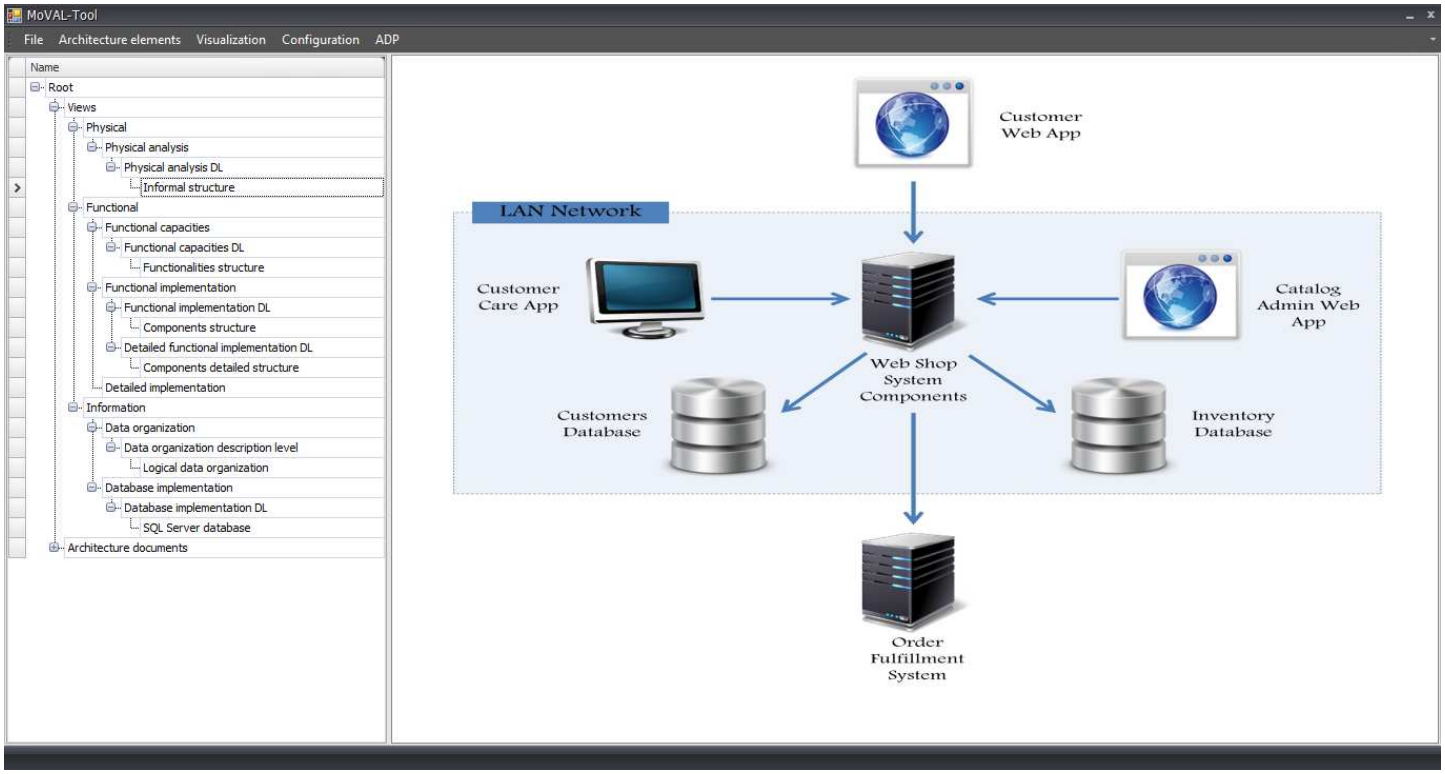


FIGURE 7.6 – L'interface graphique l'arboscence d'une architecture dans *MoVAL-Tool*.

7.4.6 Visualiser l'architecture

Visualiser l'architecture suivant une représentation hiérarchique détaillée dans laquelle on peut voir pour chaque vue de l'architecture logicielle ses niveaux de réalisation et de description et les modèles dans une hiérarchie (voir la section 4.7.2) comme illustré dans la figure 7.9.
Egalement, nous pouvons visualiser l'architecture suivant une représentation matricielle à quatre niveaux comme illustré dans la figure 7.10.

- dans le premier niveau, l'architecture est représentée sous une forme linéaire, représentant chaque vue de l'architecture dans une ligne ayant les niveaux de réalisation comme points sur cette ligne ;
- dans le deuxième niveau, les liens de correspondance $is\_$ seront ajoutés entre les différentes vues de l'architecture ;
- dans le troisième niveau, les niveaux de réalisation de chaque vue seront replacés en respectant les correspondances entre les vues ;
- finalement dans le quatrième niveau, les liens entre les vues et les niveaux de réalisation seront ajoutés sur la matrice (voir la section 4.7.3).

Le niveau de représentation est choisi selon les cases cochées à gauche de l'interface spécifiant si la matrice est ordonnée, les liens de correspondance sont représentés, les liens dérivés sont représentés et si les labels sémantiques des liens sont illustrés.

7.4.7 Utiliser le "*ADP Wizard*"

Ouvrir le "*ADP Wizard*" qui permet à l'architecte logiciel de voir sa position dans le processus de définition associé à *MoVAL*, et qui lui donne pour chaque activité des traces (*i.e. hints*), des recommandations et des meilleures pratiques (*i.e. best practices*) pour accomplir l'activité. En plus, cet "*ADP Wizard*" redirige

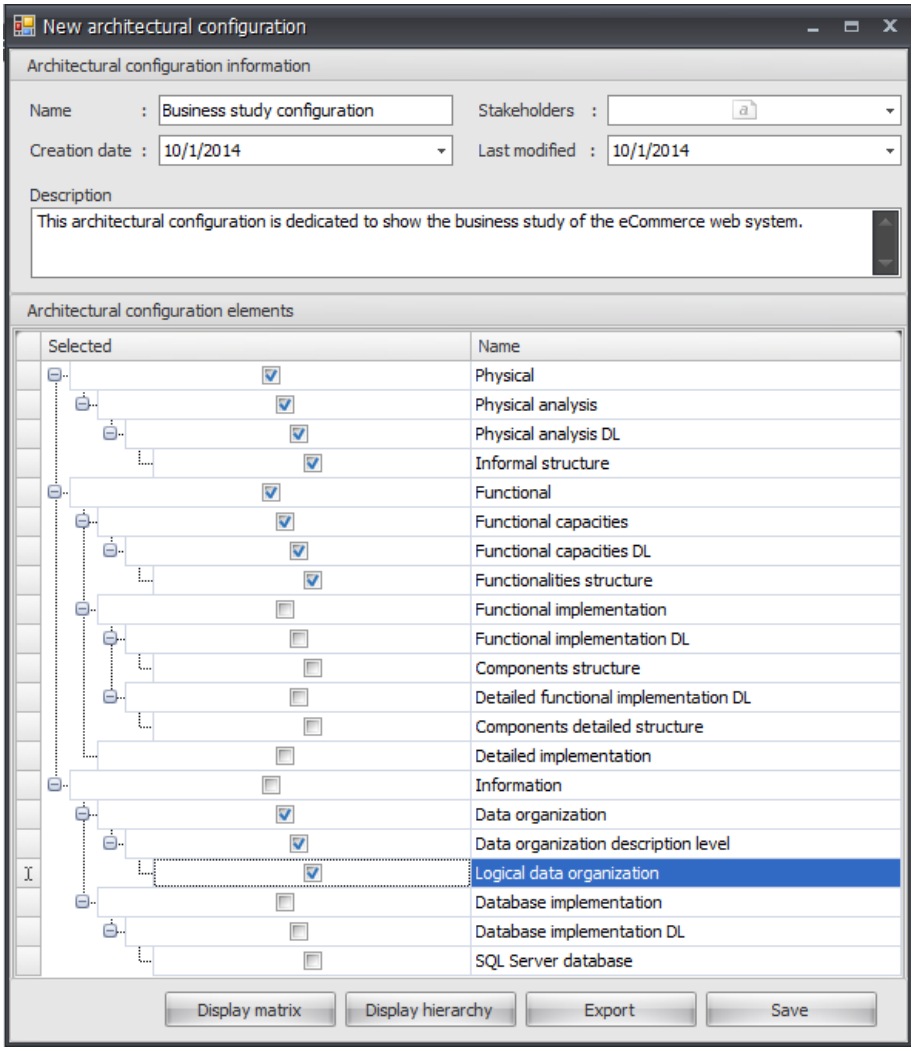


FIGURE 7.7 – L'interface de création d'une configuration architecturale dans *MoVAL-Tool*.

l'architecte aux interfaces nécessaires de l'application lui permettant d'accomplir l'activité.
Ce *Wizard* commence par une introduction sur les quatre phases principales définies dans *MoVAL-ADP*, comme illustré dans la figure 7.11. Après, la première activité par laquelle nous devons commencer est la définition des différentes itérations associées à chaque phase du processus et la création pour chacune de ces itérations un document définissant son but et son rôle. La figure 7.12 illustre l'activité de planification du processus de définition d'une architecture logicielle dans *MoVAL-Tool*.
Ensuite, le *ADP Wizard* guide l'architecte à passer par les différentes activités associées à chaque phase de l'*ADP* selon le plan défini dans la première activité en illustrant toujours la position actuelle dans le processus de définition à travers une barre de progression qui se trouve toujours en haut de l'interface de chaque activité. Ainsi, il guide l'architecte à passer par les activités de la phase inception, puis recommencer à passer par ces activités pour chaque itération supplémentaire de cette phase, et ainsi de suite pour les autres phases du processus de définition.
Voici des exemples d'activités de l'*ADP Wizard* :

- la figure 7.13 illustre l'activité spécification du contexte de la phase inception dans le *ADP Wizard*. A noter que le document créé dans cette activité et dans toute autre activité ayant un document en sortie sera accessible depuis la représentation arborescente de l'architecture (voir section 7.4.4) ;
- la figure 7.14 illustre l'activité de définition du catalogue de points de vue adopté dans l'architecture logicielle en construction ;
- la figure 7.15 illustre l'activité de construction d'une architecture candidate.

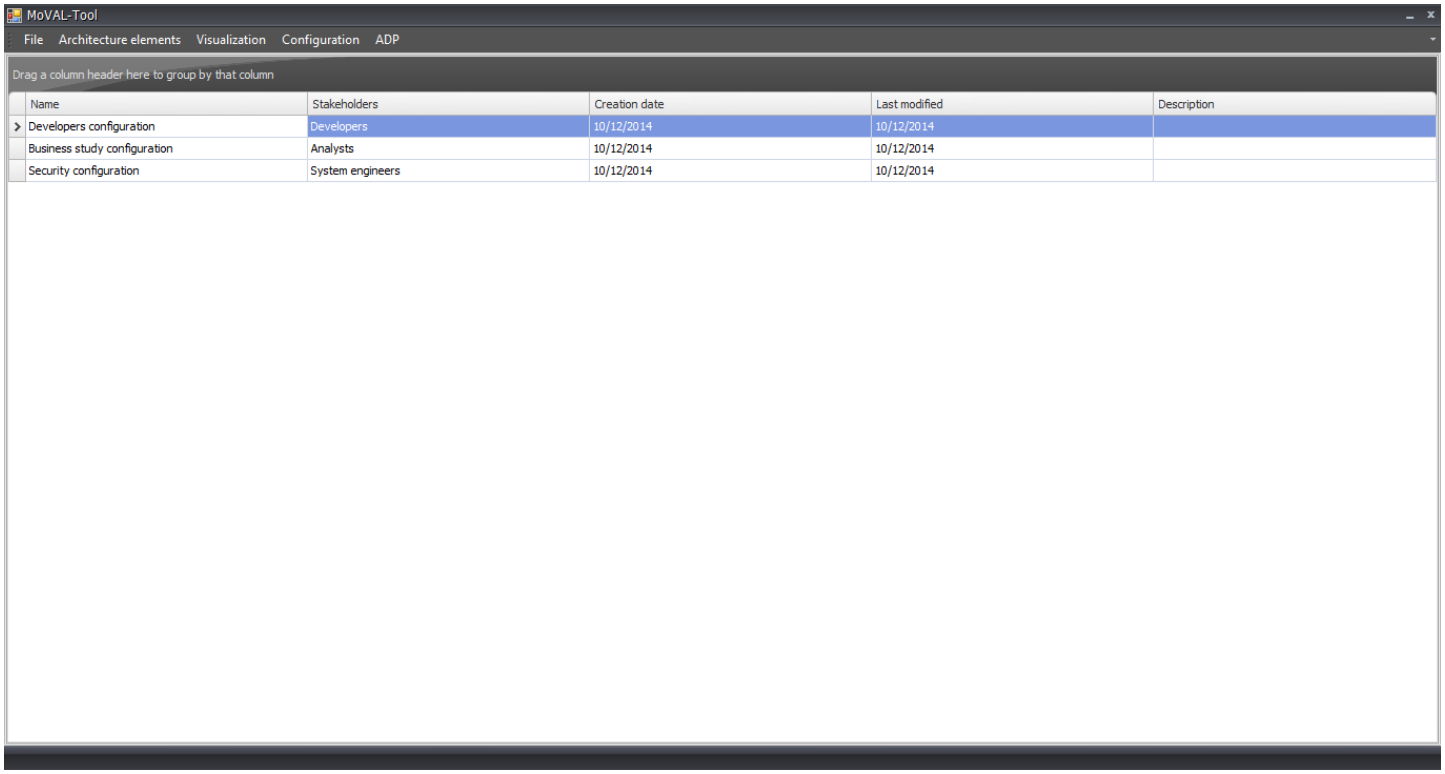


FIGURE 7.8 – La grille des configurations architecturales dans *MoVAL-Tool*.

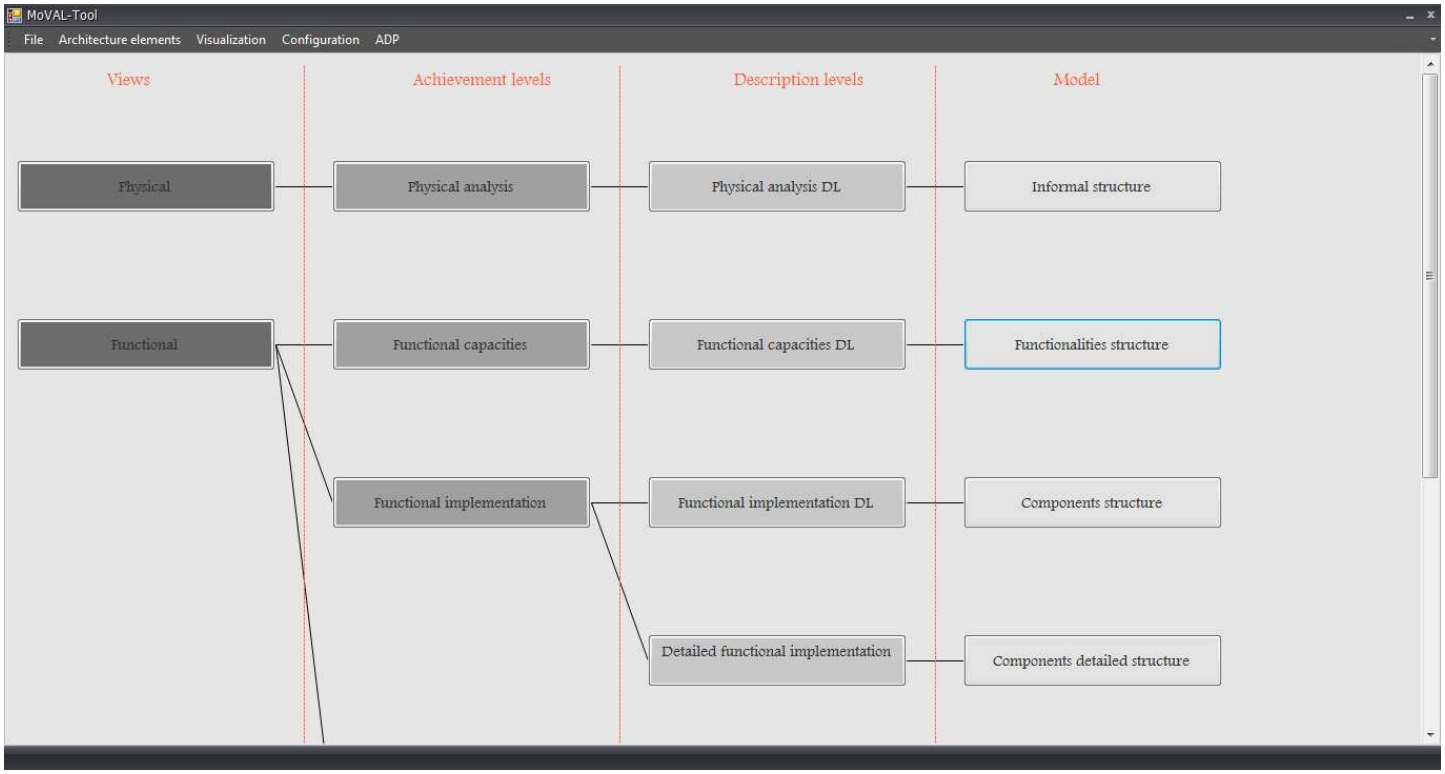


FIGURE 7.9 – La représentation hiérarchique détaillée d'une architecture logicielle dans *MoVAL-Tool*.

A noter que le *ADP Wizard* préserve l'état de construction de l'architecture même quand l'application *MoVAL-Tool* est fermée. Ainsi, il commence toujours depuis la dernière activité atteinte dans la dernière modification de l'architecture.

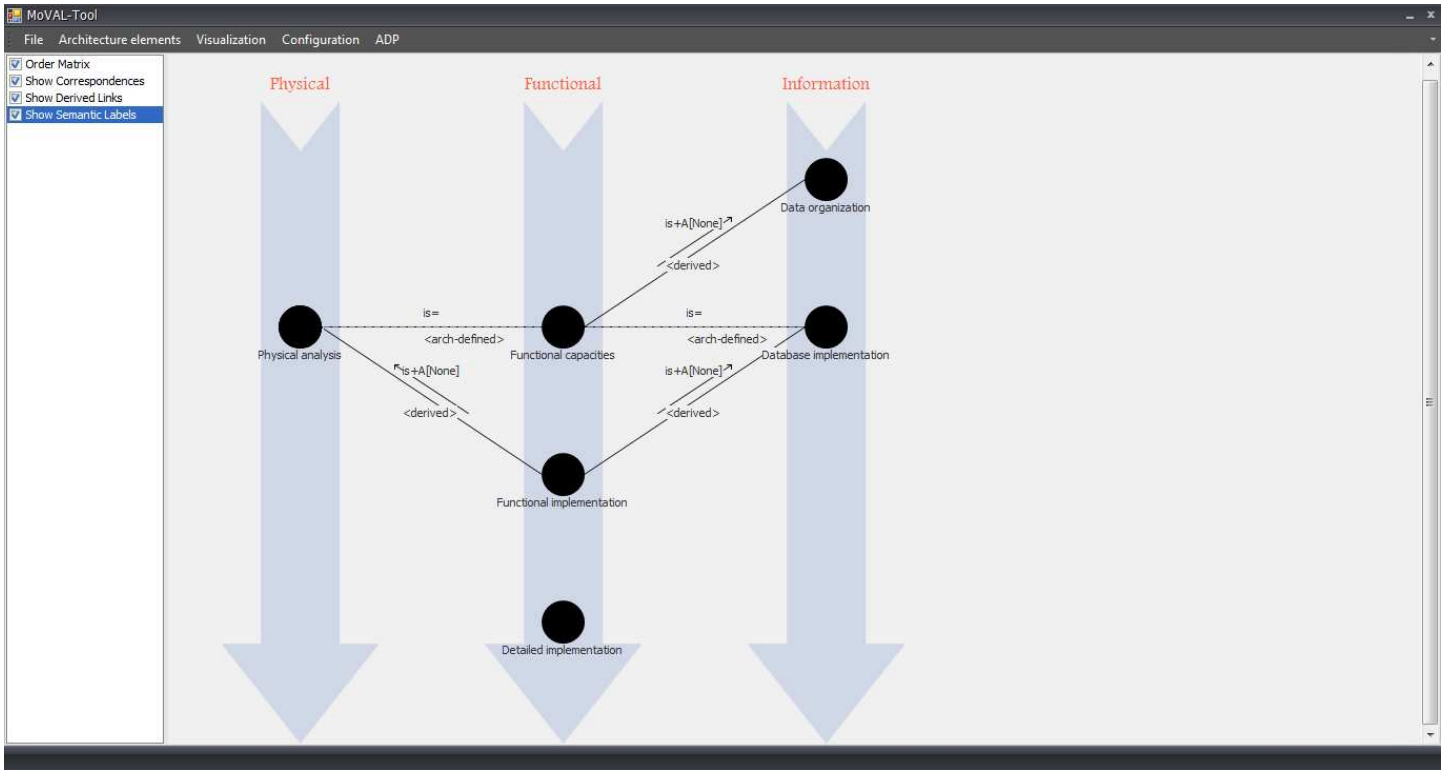


FIGURE 7.10 – La représentation matricielle d'une architecture logicielle dans *MoVAL-Tool*.

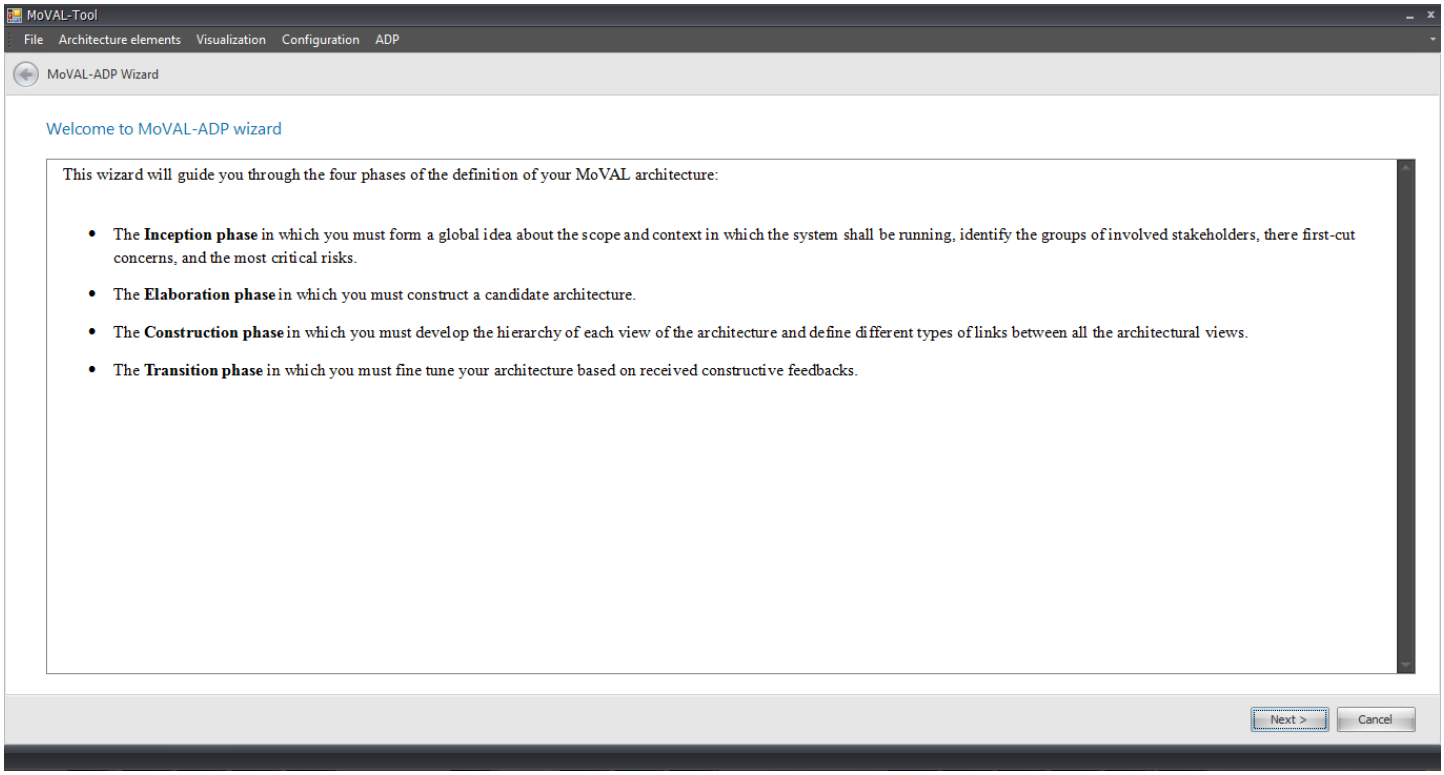


FIGURE 7.11 – L'introduction du *Wizard ADP* sur les différentes phases de *MoVAL-ADP*.

7.5 Conclusion

Dans ce chapitre, nous avons présenté les profiles UML et les langages spécifiques aux domaines (*DSL*) afin d'effectuer un choix d'une approche d'implémentation à la base des profiles UML, les *DSL* ou une application autonome. Après, nous avons choisi l'approche d'implémentation d'une application autonome

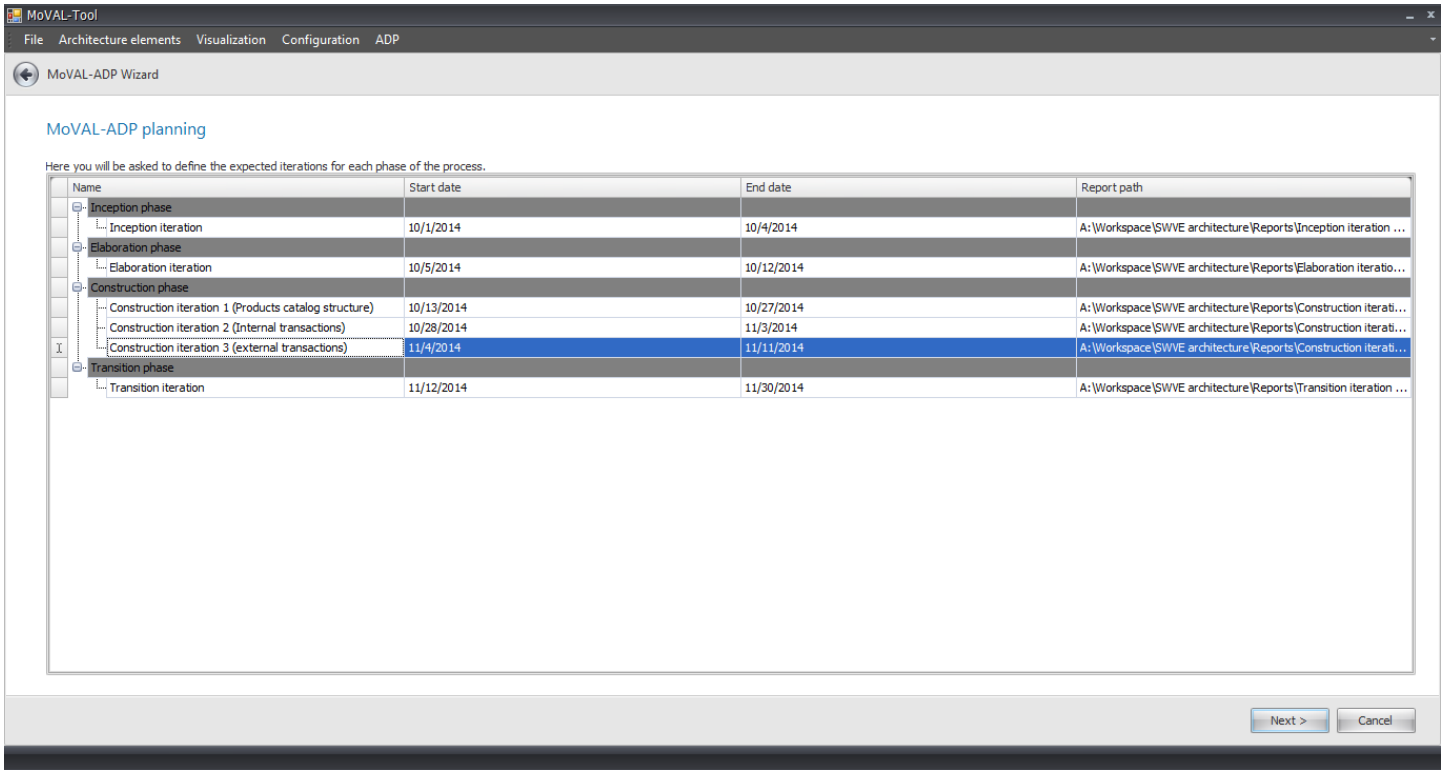


FIGURE 7.12 – L'activité de plannification du processus de définition d'une architecture dans *MoVAL-Tool*.

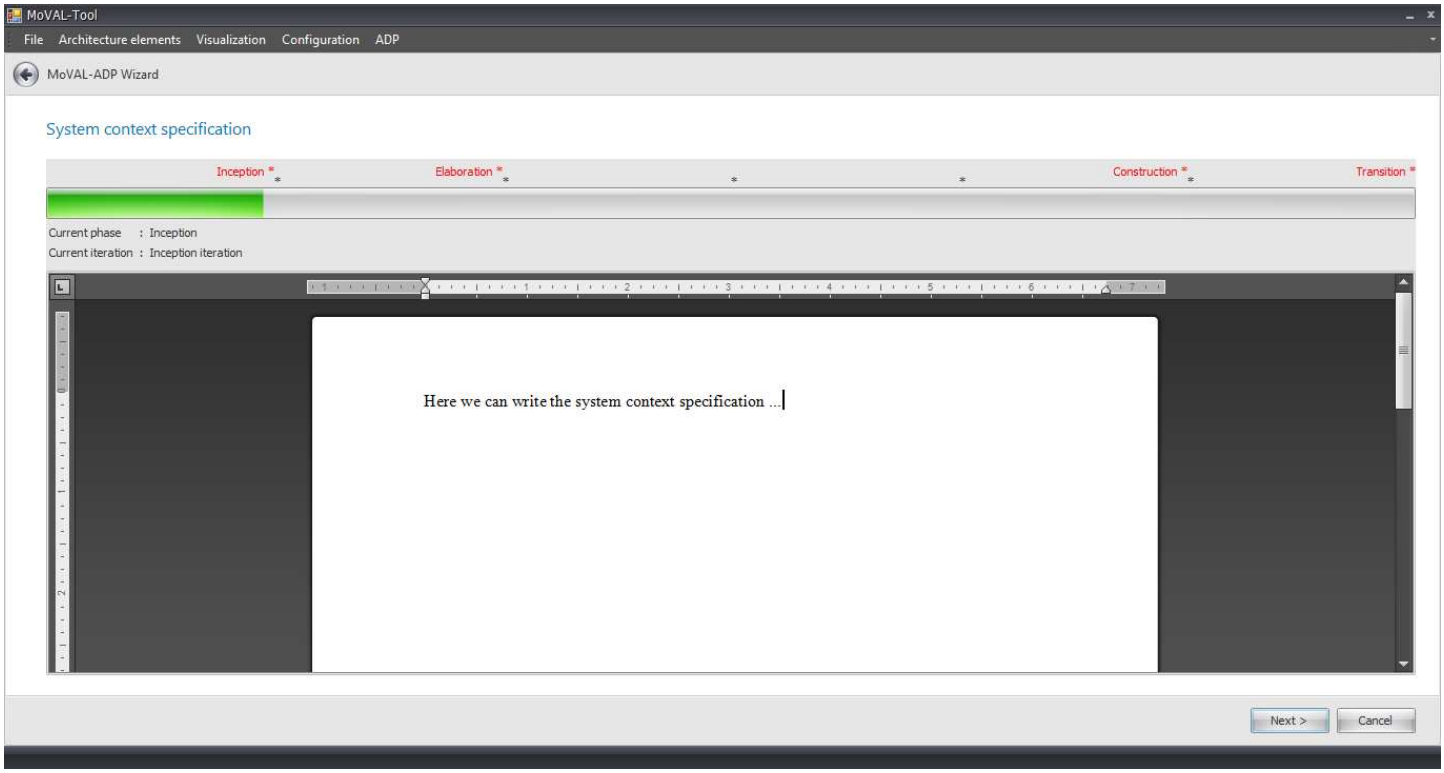


FIGURE 7.13 – L'activité de spécification du contexte d'un système du *ADP Wizard*.

suite à une petite étude comparative entre les trois approches.
Ensuite, nous avons présenté les fonctionnalités de *MoVAL-Tool*, un outil de construction d'architectures logicielles selon l'approche *MoVAL* qui a été présentée dans des chapitres précédents de cette thèse.

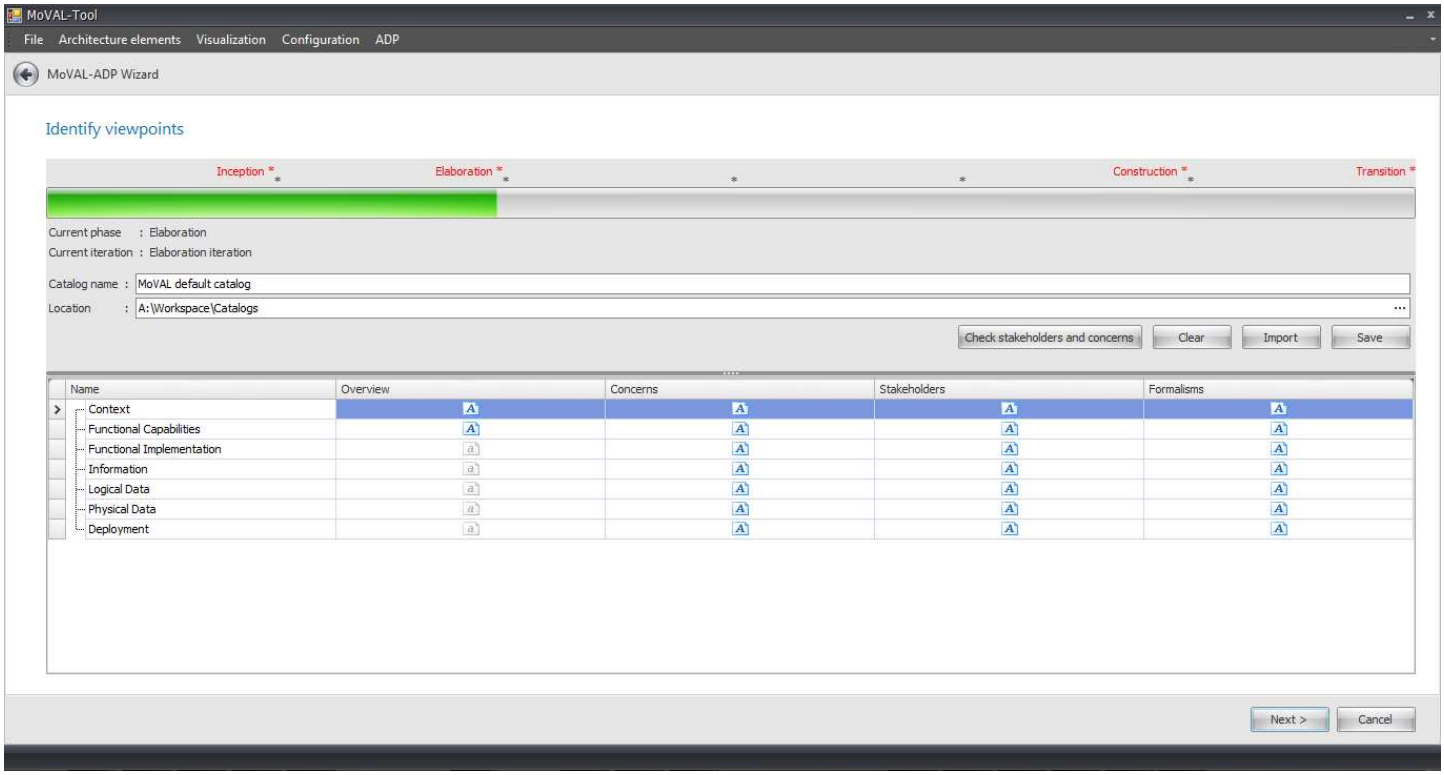


FIGURE 7.14 – L'activité de sélection ou de définition du catalogue de point de vue du *ADP Wizard*.

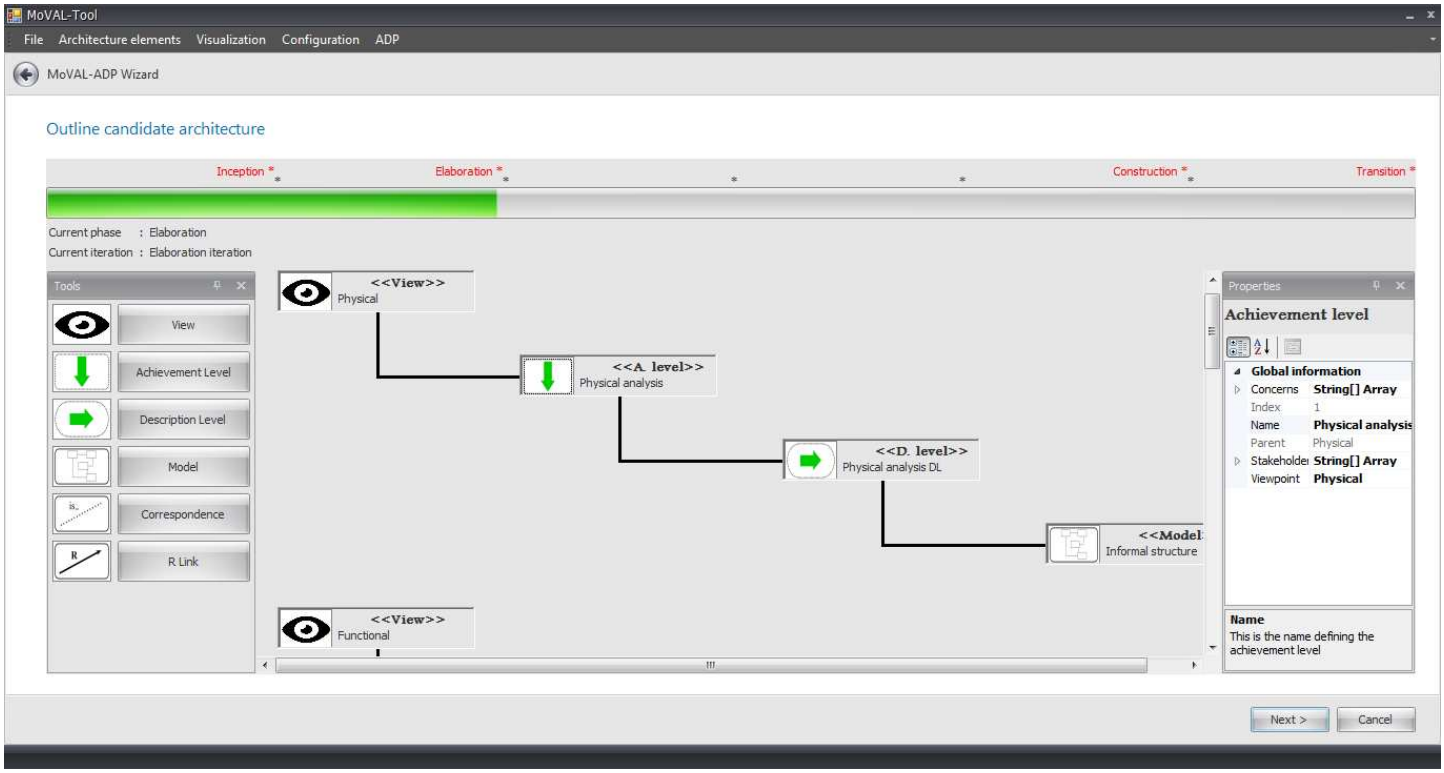


FIGURE 7.15 – L'activité de construction d'une architecture candidate du *ADP Wizard*.

8

Conclusion et perspectives

Dans cette thèse nous avons traité différents aspects liés à la définition d'une architecture logicielle selon plusieurs points de vue et plusieurs niveaux d'abstraction. Nous avons proposé l'approche d'architecture logicielle multipoints de vue *MoVAL*. Cette approche permet la définition d'une architecture logicielle de plusieurs vues et de définir une hiérarchie de plusieurs niveaux d'abstraction pour chacune de ces vues. Egalement, cette approche fournit des moyens pour spécifier des liens entre les différentes vues et niveaux de hiérarchie de l'architecture logicielle. Nous ne reviendrons pas ici sur la situation de nos travaux par rapport à des approches similaires. Cette situation ayant déjà été faite dans le chapitre 4.

8.1 Résultats

L'approche que nous avons proposé est divisée en six parties principales : (1) la structure et les éléments de base de *MoVAL*, (2) un catalogue général de points de vue pour l'application de cette approche sur des systèmes pratiques, (3) l'étude de la cohérence d'une architecture *MoVAL*, (4) une formalisation mathématique des éléments de l'approche, (5) un processus de définition d'une architecture logicielle dans *MoVAL*, et finalement (6) un prototype d'implémentation d'un outil permettant l'application de *MoVAL* dans un contexte approprié.

1. la proposition d'une approche d'architecture logicielle dénotée *MoVAL*
   Selon l'approche *MoVAL*, une architecture logicielle est composée de plusieurs vues catégorisant les intérêts d'un ensemble d'intervenants et leurs visions vis-à-vis du système informatique. Chaque vue est représentée dans plusieurs niveaux de réalisation représentant la forme du système dans la phase de développement inhérente selon l'ensemble des intervenants associés à la vue. Ce niveau de réalisation est conforme à un point de vue regroupant les préoccupations qui doivent être considérées dans cette phase et les formalismes qui peuvent être utilisés pour ce but. Après, le niveau de réalisation est organisé dans plusieurs niveaux de description qui contiennent eux même les modèles et les artefacts de l'architecture qui expriment les préoccupations associées aux intervenants de la vue impliquée ;

2. un catalogue général de points de vue
   Afin de rapprocher l'approche *MoVAL* du monde pratique et de l'industrie de développement logiciel, nous avons proposé un catalogue général de points de vue qui nous semble suffisant pour documenter une architecture logicielle et qui permet une implémentation complète et précise des exigences des différents intervenants. Ce catalogue contient un ensemble de sept points de vue avec leurs préoccu-

pations et formalismes, qui sont : le point de vue *Contexte*, *Capacités Fonctionnelles*, *Implémentation Fonctionnelle*, *Information*, *Données Logiques*, *Données Physiques*, et le point de vue *Déploiement* ;

3. l'étude de la cohérence d'une architecture *MoVAL*
   En effet, la cohérence revêt deux aspects : la cohérence structurelle et la cohérence sémantique. Pour la cohérence structurelle, nous avons défini des règles assurant la cohérence intra-vue (c.à.d. la cohérence au sein d'une vue spécifique), et des règles assurant la cohérence inter-vues (c.à.d. la cohérence entre plusieurs vues de l'architecture). En plus, pour assurer la cohérence sémantique d'une architecture *MoVAL*, nous avons défini des labels sémantiques pour les liens architecturaux. Les différents niveaux d'abstraction (*i.e.* niveaux de réalisation et niveaux de description) et les modèles de l'architecture *MoVAL* sont reliés par des éléments architecturaux, appelés liens, afin d'assurer la consistance et la cohérence de l'architecture ;

4. une formalisation mathématique des éléments de l'approche
   Nous avons présenté une formalisation mathématique des différents concepts et éléments de l'approche *MoVAL*, en utilisant la logique des prédicats. Cela afin de vérifier la cohérence de toutes les définitions des différents éléments de l'approche ;

5. un processus de définition d'une architecture logicielle dans *MoVAL*
   Nous avons proposé un processus de définition d'architecture logicielle propre à *MoVAL*, dénoté *MoVAL-ADP*, qui guide l'architecte logiciel à définir tous les éléments nécessaires de l'architecture, et lui propose un plan itératif et incrémental pour cette définition. En effet, *MoVAL-ADP* est conforme au processus unifié, et donc divisé en quatre phases différentes qui sont : la phase *Inception*, *Elaboration*, *Construction*, et *Transition* ;

6. un prototype d'un outil pour *MoVAL*
   Nous avons contruit un prototype d'un outil pour *MoVAL*, offrant les fonctionnalités nécessaires pour pouvoir utiliser et bénéficier de l'approche *MoVAL*.

8.2 Limites

Bien que les résultats de l'approche *MoVAL* soient satisfaisants et répondant aux objectifs que nous nous sommes fixés, il demeure que cette approche souffre de quatre limitations principales :

- le manque d'un langage universel de définition des règles de consistance entre les différents modèles d'une architecture logicielle dans le cas d'un label sémantique de simple connexion, ce qui implique l'impossibilité d'avoir un processus automatique de vérification de la cohérence totale de l'architecture logicielle dans ce cas ;

- le manque de règles définissant l'intégration ou les relations entre les différents points de vue définies dans un catalogue donné, afin de forcer une meilleure utilisation de ce catalogue dans les architectures logicielles ;

- le processus de définition de l'architecture *MoVAL-ADP* est conforme avec le processus unifié mais il est nécessaire d'étudier son adaptabilité avec d'autres processus comme les processus agiles selon les formations des équipes de développement et leurs stratégies de travail ;

- la formalisation mathématique des éléments de base de l'approche *MoVAL*, présentée dans cette thèse, manque plus de rigueur et de précision pour qu'elle soit plus significative.

8.3 Perspectives

Les perspectives de notre travail portent, en effet sur quatre points principaux :

- premièrement, il serait intéressant de proposer des règles et des relations d'intégration entre les différents points de vue d'un catalogue donné à l'instar des relations proposées dans le livre de *Rozanski et Woods* [Rozanski and Woods, 2011] ou dans l'approche *Rational ADS* [Norris, 2004] entre les différents points de vue de ces approches. Cela afin de soutenir la tâche de sélection des vues appropriées pour un système donné, et afin de permettre l'auto-génération d'un ensemble de liens architecturaux en se basant sur les ensembles des préoccupations des points de vue utilisés et les relations qui existent entre eux. Ces liens pourront vraiment améliorer et faciliter la résolution des inconsistances au sein de l'architecture logicielle ;

- aussi au niveau de la consistance de l'architecture, nous avons à assurer une vérification automatique de la cohérence de l'architecture logicielle en se basant sur les propriétés sémantiques des liens architecturaux et surtout sur les règles de consistance définies entre les différents éléments de l'architecture. Ainsi, ces règles de consistance doivent être définies formellement en utilisant un langage formel pour l'expression des contraintes comme le langage *OCL*. Dans ce cadre, nous devrons étudier des méthodes assurant une évaluation et une quantification des liens entre les vues, niveaux d'abstraction et modèles d'une architecture logicielle à l'instar de la méthode associée à la logique floue. Egalement, nous devrons étudier l'utilisation des mécanismes d'inférences pour une génération de liens sémantiques entre les éléments de l'architecture pour assurer une meilleure consistance au niveau de cette architecture ;

- parmi les perspectives de notre travail nous citons aussi la proposition d'un langage de description d'architecture logicielle, connu en anglais sous le nom de *Architecture Description Language* (*ADL*), dédié à *MoVAL* qui permet de définir et de décrire la structure d'une architecture logicielle et ses éléments (vues, niveaux d'abstraction, modèles et liens) à travers unÂ langage textuel formel ;

- en ce qui concerne le prototype de *MoVAL-Tool*, nous devons signaler que certaines des fonctionnalités présentées dans cette approche ne sont pas encore intégrées complètement, notamment le balayage entre les différentes activités de définition de l'architecture en se basant sur le processus de définition de l'architecture *MoVAL-ADP*, présenté dans cette thèse. Egalement, nous voudrons étendre les fonctionnalités de *MoVAL-Tool* afin de permettre aux architectes logiciels de générer un projet d'implémentation pour un environnement spécifique, qu'il soit l'environnement de développement *.NET* ou *eclipse*, en sélectionnant un ensemble de modèles qu'il prévoit appropriés à la phase d'implémentation.

8.4 Les apports de notre travail

Les apports de notre travail se divisent sur trois points principaux associés aux trois objectifs fixés dans la section 4.2 :

- La réduction de la complexité des vues architecturales en les stratifiant en plusieurs niveaux de réalisation et de description, permettant à l'équipe de développement de se concentrer sur un sous-ensemble de l'ensemble des préoccupations associées à chaque vue à la fois ;
  D'une autre part, les vues architecturaux peuvent être communiquées, selon notre approche *MoVAL*, à un intervenant particulier en utilisant les modèles associés aux niveaux de réalisation et de description adaptés au connaissances techniques et fonctionnelles de cet intervenant. D'où la réduction de la complexité de compréhension et de communication de la vue avec les intervenants ;

- L’assurance de la cohérence de l’architecture logicielle en définissant des liens entre les différents niveaux de réalisation, de description, et les différents modèles de l’architecture, de manière à pouvoir détecter les inconsistances potentielles qui peuvent arriver suite à des modifications sur l’architecture, ou suite à la complexité des exigences du système ;
- La définition d’un processus de définition d’architecture logicielle guidant l’architecte à définir l’architecture d’une manière efficace en utilisant les meilleures pratiques que nous trouvons les plus précises, performantes, et optimisées.

Table des matières

Liste des tableaux

Table des figures

Bibliographie

[Andrade et al., 2004] Andrade, J., Ares, J., Garcia, R., Pazos, J., Rodriguez, S., and Silva, A. (2004). A methodological framework for viewpoint-oriented conceptual modeling. *Software Engineering, IEEE Transactions on*, 30(5) :282–294. 20, 137

[Anwar et al., 2011] Anwar, A., Dkaki, T., Ebersold, S., Coulette, B., and Nassar, M. (2011). A formal approach to model composition applied to VUML. In *Engineering of Complex Computer Systems (ICECCS), 2011 16th IEEE International Conference on*, pages 188–197. 22

[Anwar et al., 2010] Anwar, A., Ebersold, S., Coulette, B., Nassar, M., and Kriouile, A. (2010). A rule-driven approach for composing viewpoint-oriented models. *Journal of Object Technology*, 9(2) :89–114. 22

[Beck, 1999] Beck, K. (1999). Embracing change with extreme programming. *Computer*, 32(10) :70–77. 90

[Board, 1998] Board, I.-S. S. (1998). Ieee recommended practice for software requirements specifications. *IEEE-Std-830-1998*. 12

[Bobrow and Stefik, 1983] Bobrow, D. G. and Stefik, M. (1983). *The Loops Manual : Preliminary Vision*. Intelligent Systems Laboratory, Xerox Corporation. 11

[Bobrow and Winograd, 1977] Bobrow, D. G. and Winograd, T. (1977). An overview of KRL, a knowledge representation language. *Cognitive Science*, 1(1) :3–46. 11

[Booch and Rumbaugh, 1999] Booch, Grady, J. I. and Rumbaugh, J. (1999). *The unified software development process*. Addison-Wesley. 87, 88

[Bowman et al., 1995] Bowman, H., Derrick, J., and Steen, M. (1995). Some results on cross viewpoint consistency checking. In *Open Distributed Processing*, pages 399–412. Springer. 36

[Bowman et al., 2002] Bowman, H., Steen, M. W., Boiten, E. A., and Derrick, J. (2002). A formal framework for viewpoint consistency. *Formal Methods in System Design*, 21(2) :111–166. 36

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern-Oriented Software Architecture, Volume 1 : A System of Patterns*. Wiley. 30

[Carré et al., 1990] Carré, B., Dekker, L., and Geib, J.-M. (1990). Multiple and evolutive representation in the rome language, towards an integrated company information system. *TOOLS 1990*. 11

[Clements, 2003] Clements, P. (2003). *Documenting Software Architectures : Views and Beyond*. Addison-Wesley Professional. 25

[Clements et al., 2002] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., and Stafford, J. (2002). A practical method for documenting software architectures. 29, 30, 137

[Cohn, 2005] Cohn, M. (2005). *Agile estimating and planning*. Pearson Education. 91, 138

[Delugach, 1990] Delugach, H. S. (1990). Using conceptual graphs to analyze multiple views of software requirements. 17

[Delugach, 1991] Delugach, H. S. (1991). *A multiple viewed approach to software requirements*. PhD thesis, Citeseer. 17

[Delugach, 1992] Delugach, H. S. (1992). Specifying multiple-viewed software requirements with conceptual graphs. *Journal of Systems and Software*, 19(3) :207–224. 17

[Delugach, 1996] Delugach, H. S. (1996). An approach to conceptual feedback in multiple viewed software requirements modeling. In *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints 96) on SIGSOFT 96 workshops*, pages 242–246. ACM. 17, 137

[DeMarco, 1979] DeMarco, T. (1979). *Structured analysis and system specification*. Yourdon Press. 50

[Dijkman, 2006] Dijkman, R. M. (2006). *Consistency in multi-viewpoint architectural design*. PhD thesis, University of Twente. 21

[Dijkman et al., 2008] Dijkman, R. M., Quartel, D. A. C., and van Sinderen, M. J. (2008). Consistency in multi-viewpoint design of enterprise information systems. *Information and Software Technology*, 50(7) :737–752. 21, 137

[Easterbrrok and Nuseibeh, 1996] Easterbrrok, S. and Nuseibeh, B. (1996). Using viewpoints for inconsistency management. *Software Engineering Journal*, 11(1) :31–43. 16

[El Asri et al., 2006] El Asri, B., Nassar, M., Coulette, B., and Kriouile, A. (2006). Architecture d'assemblage dynamique de composants multivues dans VUML. In *INFORSID*, pages 943–958. 22

[Erman and Lesser, 1975] Erman, L. D. and Lesser, V. R. (1975). A multi-level organization for problem solving using many, diverse, cooperating sources of knowledge. Technical report, DTIC Document. 11

[ExecutiveBrief, 2011] ExecutiveBrief (2011). 2011 software development trends survey results. *2011 Software Development Trends*. 90

[Finkelstein and Fuks, 1989] Finkelstein, A. and Fuks, H. (1989). Multiparty specification. In *ACM SIGSOFT Software Engineering Notes*, volume 14, pages 185–195. 14, 15, 137

[Finkelstein et al., 1993] Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B. (1993). Inconsistency handling in multi-perspective specifications. In *Software Engineering-ESEC 93*, pages 84–99. Springer. 14

[Finkelstein et al., 1991] Finkelstein, A., Goedicke, M., Kramer, J., and Niskier, C. (1991). ViewPoint oriented software development : Methods and viewpoints in requirements engineering. In Bergstra, J. A. and Feijs, L. M. G., editors, *Algebraic Methods II : Theory, Tools and Applications*, number 490 in Lecture Notes in Computer Science, pages 29–54. Springer Berlin Heidelberg. 16, 137

[Fowler and Highsmith, 2001] Fowler, M. and Highsmith, J. (2001). The agile manifesto. *Software Development*, 9(8) :28–35. 87

[Graversen, 2006] Graversen, K. B. (2006). *The nature of roles A taxonomic analysis of roles as a language construct*. PhD thesis, IT University of Copenhagen. 24

[Grundy et al., 1998] Grundy, J., Hosking, J., and Mugridge, W. B. (1998). Inconsistency management for multiple-view software development environments. *Software Engineering, IEEE Transactions on*, 24(11) :960–981. 25

[Harrison and Ossher, 1993] Harrison, W. and Ossher, H. (1993). *Subject-oriented programming : a critique of pure objects*, volume 28. ACM. 23

[Hazzan and Kramer, 2007] Hazzan, O. and Kramer, J. (2007). Abstraction in computer science & software engineering : A pedagogical perspective. *Frontier Journal*, 4(1) :6–14. 58

[Hilliard, 2007] Hilliard, R. (2007). Using aspects in architectural description. In *Early Aspects : Current Challenges and Future Directions*, pages 139–154. Springer. 36

[Hilliard et al., 2012] Hilliard, R., Malavolta, I., Muccini, H., and Pelliccione, P. (2012). On the composition and reuse of viewpoints across architecture frameworks. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 131–140. IEEE. 37

[Hirschfeld et al., 2008] Hirschfeld, R., Costanza, P., and Nierstrasz, O. (2008). Context-oriented programming. *Journal of Object Technology*, 7(3). 25

[ISO/IEC/IEEE, 2011] ISO/IEC/IEEE (2011). Systems and software engineering – architecture description. *ISO/IEC/IEEE 42010 :2011(E) (Revision of ISO/IEC 42010 :2007 and IEEE Std 1471-2000)*. 7, 28, 29, 54, 56, 74, 78, 137

[Kheir et al., 2013] Kheir, A., Naja, H., and Oussalah, M. (2013). Hierarchical multi-views software architecture. In *ICSEA 2013 : The Eighth International Conference on Software Engineering Advances*, pages 478–483, Italy. 78

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. *ECOOP 97-Object-Oriented Programming*, pages 220–242. 24

[Klug and Tsichritzis, 1977] Klug, A. C. and Tsichritzis, D. (1977). Multiple view support within the ANSI/SPARC framework. In *Proceedings of the Third International Conference on Very Large Data Bases - Volume 3*, VLDB '77, pages 477–488, Tokyo, Japan. VLDB Endowment. 11

[Kriouile, 1995] Kriouile, A. (1995). VBOOM, une méthode orientée objet d'analyse et de conception par points de vue. *Doctorat thesis, University of Mohamed V, Rabat, Maroc*. 20

[Kruchten, 1995] Kruchten, P. (1995). The 4+ 1 view model of architecture. *Software, IEEE*, 12(6) :42 – 50. 26, 27, 137

[Laddad, 2009] Laddad, R. (2009). *AspectJ in Action : Enterprise AOP with Spring Applications*. Manning Publications Co., Greenwich, CT, USA, 2nd edition. 23

[Larousse, 1995] Larousse (1995). *GRAND LAROUSSE POUR TOUS*. Larousse. 5

[Majumdar and Bhattacharya, 2010] Majumdar, D. and Bhattacharya, S. (2010). Aspect oriented requirements engineering : A theme based vector-orientation model. *Infocomp Journal of Computer Science*. 24

[Marcaillou et al., 1994] Marcaillou, S., Coulette, B., and Kriouile, A. (1994). Visibility : A new relationship for complex system modelling. *TOOLS USA 94*, pages 1–5. 20

[Martin, 1991] Martin, J. (1991). *Rapid Application Development*. Macmillan Publishing Co., Inc., Indianapolis, IN, USA. 92

[May, 2005] May, N. (2005). A survey of software architecture viewpoint models. In *Proceedings of the Sixth Australasian Workshop on Software and System Architectures*, pages 13–24. Citeseer. 26, 27, 28, 34, 137

[Mcheick et al., 2006] Mcheick, H., Mili, H., Sadou, S., and El-Kharraz, A. (2006). A comparison of aspect oriented software development techniques for distributed applications. *IADIS press*, pages 324–333. 24

[Medvidovic et al., 1996] Medvidovic, N., Taylor, R. N., and Whitehead Jr, E. J. (1996). Formal modeling of software architectures at multiple levels of abstraction. *ejw*, 714 :824–2776. 58

[Mili et al., 1999] Mili, H., Dargham, J., Mili, A., Cherkaoui, O., and Godin, R. (1999). View programming for decentralized development of OO programs. In *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 30. Proceedings*, pages 210–221. 24

[Miller and Mukerji, 2003] Miller, J. and Mukerji, J. (2003). MDA guide version 1.0. 1. *Object Management Group*, 234 :51. 54, 58

[Minksy, 1975] Minksy, M. (1975). A framework for representing knowledge. *The Psychology of Computer Vision, McGraw-Hill*, pages 211–277. 11

[Monperrus et al., 2009] Monperrus, M., Beugnard, A., Champeau, J., et al. (2009). A definition of "abstraction level" for metamodels. In *ECBS*, pages 315–320. 58

[Mullery, 1979] Mullery, G. P. (1979). CORE-a method for controlled requirement specification. In *Proceedings of the 4th international conference on Software engineering*, pages 126–135. 13, 14, 137

[Naja, 1998] Naja, H. (1998). La représentation multiple d'objets pour l'ingénierie. *Revue l'Objet : logiciel, bases de donnÃ©es, rÃ©seaux*, 4(2) :173–191. 11, 19

[Nassar, 2003] Nassar, M. (2003). VUML : a viewpoint oriented UML extension. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 373–376. 21

[Nassar et al., 2009] Nassar, M., Anwar, A., Ebersold, S., Elasri, B., Coulette, B., and Kriouile, A. (2009). Code generation in VUML profile : A model driven approach. In *Computer Systems and Applications, 2009. AICCSA 2009. IEEE/ACS International Conference on*, pages 412–419. 22

[Nassar et al., 2005] Nassar, M., Coulette, B., Guiochet, J., Ebersold, S., El Asri, B., Crégut, X., and Kriouile, A. (2005). Vers un profil UML pour la conception de composants multivues. *L'OBJET*, 11(4) :83–113. 22

[Nassar et al., 2004] Nassar, M., El Asri, B., Coulette, B., and Kriouile, A. (2004). Une approche UML de composants multivues. In *Workshop Objets-Composants-Modèles dans les Systèmes d'Information (OCM-SI 2004). Biarritz, France*, volume 25. 21, 22, 137

[Naur and Randell, 1969] Naur, P. and Randell, B. (1969). Software engineering : Report of a conference sponsored by the NATO science committee, garmisch, germany, 7-11 oct. 1968, brussels, scientific affairs division, NATO. 5

[Niskier et al., 1989] Niskier, C., Maibaum, T., and Schwabe, D. (1989). A look through PRISMA : towards pluralistic knowledge-based environments for software specification acquisition. In *Proceedings of the 5th International Workshop on Software Specification and Design*, IWSSD 89, pages 128–136, New York, NY, USA. ACM. 16

[Norris, 2004] Norris, D. (2004). Communicating complex architectures with UML and the rational ADS. In *Proceedings of the IBM Rational Software Development User Conference*. 27, 129

[Nuseibeh et al., 1994] Nuseibeh, B., Kramer, J., and Finkelstein, A. (1994). A framework for expressing the relationships between multiple views in requirements specification. *Software Engineering, IEEE Transactions on*, 20(10) :760–773. 16

[OMG, 2011] OMG (2011). Omg unified modeling languagetm (omg uml), infrastructure. In *OMG's industry-standard environment*. 20

[OMG, 2013] OMG (2013). OMG meta object facility (MOF) core specification. In *OMG's industry-standard environment*. 54, 78

[Ossher et al., 1995] Ossher, H., Kaplan, M., Harrison, W., Katz, A., and Kruskal, V. (1995). Subject-oriented composition rules. In *ACM SIGPLAN Notices*, volume 30, pages 235–250. 23

[Oussalah and al., 2005] Oussalah, M. and al. (2005). *Ingénierie des composants : Concepts, Techniques et Outils*. Vuibert. 23

[Oussalah and al., 2014] Oussalah, M. and al. (2014). *Architectures logicielles : Principes, techniques et outils*. Hermes Sciences-Lavoisier, Paris. 26

[Parnas, 1971] Parnas, D. L. (1971). Information distribution aspects of design methodology. 25

[Perry and Wolf, 1992] Perry, D. E. and Wolf, A. L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4) :40–52. 25

[Raymond, 1995] Raymond, K. (1995). Reference model of open distributed processing (RM-ODP) : introduction. In *Open Distributed Processing*, pages 3–14. Springer. 35

[Raymond and Armstrong, 1995] Raymond, K. and Armstrong, L. (1995). *Open Distributed Processing*. Springer. 35

[Regnell et al., 1996] Regnell, B., Andersson, M., and Bergstrand, J. (1996). A hierarchical use case model with graphical representation. In *Engineering of Computer-Based Systems, 1996. Proceedings., IEEE Symposium and Workshop on*, pages 270–277. 58

[Robinson, 1990] Robinson, W. N. (1990). Negotiation behavior during requirement specification. In *Software Engineering, 1990. Proceedings., 12th International Conference on*, pages 268–276. 14

[Ross and Schoman Jr, 1977] Ross, D. T. and Schoman Jr, K. E. (1977). Structured analysis for requirements definition. *Software Engineering, IEEE Transactions on*, (1) :6–15. 12

[Rozanski and Woods, 2011] Rozanski, N. and Woods, E. (2011). *Software systems architecture : working with stakeholders using viewpoints and perspectives*. Addison-Wesley. 6, 31, 32, 33, 52, 87, 88, 89, 129, 137, 138

[Rumbaugh et al., 1991] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorensen, W. E., et al. (1991). *Object-oriented modeling and design*, volume 199. Prentice-hall Englewood Cliffs, NJ. 23

[Salo and Abrahamsson, 2008] Salo, O. and Abrahamsson, P. (2008). Agile methods in european embedded software development organisations : a survey on the actual use and usefulness of extreme programming and scrum. *Software, IET*, 2(1) :58–64. 90

[Sampaio do Prado Leite and Freeman, 1991] Sampaio do Prado Leite, J. C. and Freeman, P. A. (1991). Requirements validation through viewpoint resolution. *Software Engineering, IEEE Transactions on*, 17(12) :1253–1269. 16

[Schwaber and Beedle, 2002] Schwaber, K. and Beedle, M. (2002). *Agile software development with scrum*. Series in agile software development. Prentice Hall. 90

[Shiling and Sweeney, 1989] Shiling, J. J. and Sweeney, P. F. (1989). *Three steps to views : extending the object-oriented paradigm*, volume 24. ACM. 24

[Smolander, 2002] Smolander, K. (2002). What is included in software architecture ? a case study in three software organizations. In *Engineering of Computer-Based Systems, 2002. Proceedings. Ninth Annual IEEE International Conference and Workshop on the*, pages 131–138. IEEE. 26

[Sommerville and Sawyer, 1997] Sommerville, I. and Sawyer, P. (1997). Viewpoints : principles, problems and a practical approach to requirements engineering. *Annals of Software Engineering*, 3(1) :101–130. 17, 18, 19, 137

[Soni et al., 1995] Soni, D., Nord, R. L., and Hofmeister, C. (1995). Software architecture in industrial applications. In *Software Engineering, 1995. ICSE 1995. 17th International Conference on*, pages 196–196. IEEE. 33

[Sowa and Zachman, 1992] Sowa, J. F. and Zachman, J. A. (1992). Extending and formalizing the framework for information systems architecture. *IBM systems journal*, 31(3) :590–616. 34

[Technology, 2008] Technology, M. (2008). Mdg technology for zachman framework user guide. Technical report, Zachman Entreprise. 34

[van Heesch et al., 2012] van Heesch, U., Avgeriou, P., and Hilliard, R. (2012). A documentation framework for architecture decisions. *Journal of Systems and Software*, 85(4) :795–820. 37

[Vijayasarathy and Turk, 2008] Vijayasarathy, L. and Turk, D. (2008). Agile software development : A survey of early adopters. *Journal of Information Technology Management*, 19(2) :1–8. 90

[Zachman, 1987] Zachman, J. A. (1987). A framework for information systems architecture. *IBM systems journal*, 26(3) :276–292. 34

[Zachman, 1996] Zachman, J. A. (1996). Concepts of the framework for enterprise architecture. *Zachman International*. 34

Les publications de l'équipe

[1] Ahmad Kheir , Hala Naja, and Mourad Oussalah. *An Iterative Architectural Description Process Based On Views and Abstraction Levels*. Journal of Systems and Software (JSS), Elsevier, (Submitted), 2014.

[2] Ahmad Kheir , Hala Naja, and Mourad Oussalah. *A comparative study on Views in Software Engineering & a contribution to overcome some limitations in Software Architecture Field*. InfoComp Journal, 13(1), 26-37, 2014.

[3] Ahmad Kheir, Hala Naja, Mourad Oussalah, Kifah Tout. *MoVAL : Towards a Multi-views/Multi-hierarchy Software Architecture*. in : 20th LAAS International Science Conference, Beirut, Lebanon, March 2014.

[4] Ahmad Kheir , Hala Naja, and Mourad Oussalah. *Multihierarchy/Multiview Software Architectures (chap. 3)*. in : Software Architecture 1. Wiley, p. 83-118, April 2014.

[5] Ahmad Kheir , Hala Naja, and Mourad Oussalah. *Hierarchical Multi-Views Software Architecture*. In the The Eighth International Conference on Software Engineering Advances (ICSEA 2013), Italy, 2013.

[6] Ahmad Kheir, Hala Naja, Mourad Oussalah, and Kifah Tout. *From viewpoints and abstraction levels in software engineering towards multi- Viewpoints/Multi-Hierarchy in software architecture*. In 8th International Conference on Software Engineering and Applications (ICSOFT-EA 2013), Iceland, July 2013.

[7] Ahmad Kheir, Hala Naja, Mourad Oussalah, and Kifah Tout. *Overview of an approach describing multi-Views/Multi-Abstraction levels software architecture*. In 8th International conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2013), France, July 2013.

Thèse de Doctorat

Ahmad KHEIR

MoVAL : Modélisation multipoints de vue /multi-granularités d'architectures logicielles

Multi-viewpoints/multi-granularity software architectures modeling

Résumé

Le travail conduit dans cette thèse a pour but de développer des architectures logicielles hiérarchisées et multipoints de vue réduisant les différents types de complexités qui peuvent avoir lieu à différents niveaux du processus de développement logiciel.
Egalement, nous avons développé *MoVAL*, qui est une approche d'architecture logicielle à base de modèles, vues, et niveaux d'abstraction. Cette approche se base sur la construction d'architecture logicielle multipoints de vue, et la décomposition de chacune de ces vues en différents niveaux d'abstraction de plusieurs types.
En fait, l'approche *MoVAL* étend le standard *IEEE 42010* et est en plus conforme à d'autres standards importants en génie logiciel, comme le standard *MOF* (*Meta-Object Facility*).
Une vue dans *MoVAL* est une représentation d'un système intégrant un ensemble d'aspects reliés au processus de développement, et associés à une catégorie spécifique d'intervenants ou un groupe de catégories d'intervenants. Chaque vue est définie selon plusieurs niveaux d'abstraction de différents types : les niveaux de réalisation et les niveaux de description. A noter que les vues et les niveaux d'abstraction d'une architecture *MoVAL* sont liés entre eux par des éléments architecturaux formels permettant d'assurer la consistance de l'architecture.
Enfin, afin de profiter de l'approche *MoVAL* d'une manière efficace, nous avons défini une méthodologie spécifique pour guider l'architecte pendant le développement de son architecture. Nous avons ainsi développé un processus de définition d'architecture spécifique à *MoVAL* et conforme avec le processus unifié (*UP*).

Mots clés

Architecture logicielle, Point de vue, Hiérarchie, Abstraction.

Abstract

The work conducted in this thesis aims to build hierarchical multi-viewpoints software architectures that reduce different types of complexity found in different stages of the software development process.
Hence, we developed *MoVAL* a Model, View, and Abstraction Level based software architecture approach that is based on the construction of multi-views architectures, and the decomposition of each view to multiple abstraction levels of several types.
Actually, MoVAL extends the *IEEE 42010* standard and also complies with other important standards in software engineering, like the *Meta-Object Facility* (*MOF*).
A view in *MoVAL* is a representation of the system considering a set of the development process' aspects, and some problems associated to a specific category of stakeholders or a group of categories of stakeholders. This view is defined in multiple levels of abstraction of different types: the achievement levels and description levels. Note that views and abstraction levels of a MoVAL architecture are linked together via formal architectural element called links, in order to ensure the consistency of that architecture.
Now, In order to benefit from *MoVAL* approach, it was crucial to define a methodology that guides the architect while developing his architecture. For this reason, a MoVAL specific architecture definition process (*ADP*) that complies with the *Unified Process* (*UP*) was developed.

Key Words

Software architecture, Viewpoint, Hierarchy, Abstraction.